INSTITUTO POLITÉCNICO DE BRAGANÇA

ESCOLA SUPERIOR DE TECNOLOGIA E DE GESTÃO

Engenharia Electrotécnica

Microprocessadores

2ºano - 1ºsemestre

Microprocessador 8085

Programação em Linguagem Assembly

Ângelo César
Web Page: http://www.ipb.pt/~aacesar
E-Mail: aacesar@ipb.pt

Bragança, Setembro de 2002

ÍNDICE

1. ESTRUTURA BÁSICA DO MICROCOMPUTADOR

O microcomputador é composto por várias unidades e a sua estrutura básica está exposta na figura seguinte:

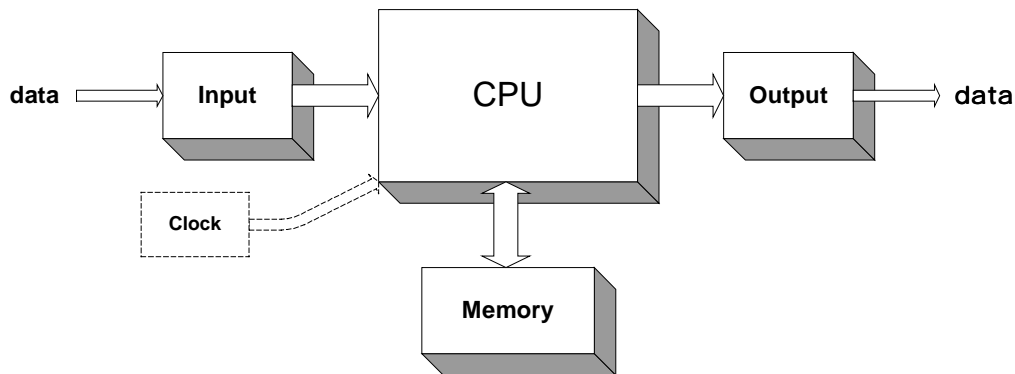



Figura 1.1 – **Estrutura básica do microcomputador**

A unidade de processamento central, conhecida pela sigla inglesa CPU – *Central Processing Unit* –, é a unidade onde são processados os dados de natureza binária, vindos da unidade de entrada (*Input*). A unidade de memória (*Memory*) serve para armazenar os dados que serão posteriormente utilizados pelo CPU.

O CPU processa os dados entrados, juntamente com os dados guardados em memória, de acordo com o programa previamente definido e a ser executado pelo CPU. Os resultados deste processo são armazenados na memória ou transferidos para a unidade de saída (*Output*).

Os dados que atravessam as unidades do microcomputador, assim como os programas e os dados armazenados em memória, são de natureza binária.

1.1. UNIDADE DE ENTRADA

Os dados são introduzidos no microcomputador através da unidade de entrada, utilizando diversos meios: pressão sobre uma tecla, tensões eléctricas diferentes, níveis de temperatura diferentes, etc.

O teclado corresponde à unidade de entrada do microcomputador; cada tecla tem associado um valor numérico específico. Ao pressionar uma tecla o seu valor é transferido através da unidade de entrada do teclado para o CPU.

O CPU recebe o valor numérico, processa-o e transfere o valor processado para a memória, para a unidade de saída ou utiliza-o num novo cálculo. O processo de

transferência do resultado final é feito de acordo com o programa a ser executado. Neste programa estão também definidas as acções a realizar para que o CPU receba os dados da unidade de entrada.

Exemplo de aplicação:
O microcomputador controla o funcionamento do sistema de ar condicionado através da temperatura do ar ambiente. Um sensor de temperatura permite converter a informação da temperatura num sinal eléctrico, geralmente uma tensão. A variação da tensão eléctrica, proporcional à variação da temperatura ambiental, é transmitida ao micro-computador.

A informação que o microcomputador recebe, depois de convertida pela unidade de entrada, é uma grandeza eléctrica binária. Os dados são transferidos para o processador que de seguida os processa.

Depois de processados os dados são transferidos para a unidade de saída. A informação que sai do microcomputador irá operar o sistema de refrigeração ou o sistema de aquecimento, dependendo do valor debitado à saída, relativamente aos valores de referência considerados, quer o valor máximo quer o valor mínimo.

1.2. UNIDADE DE SAÍDA

A unidade de saída permite converter os dados binários enviados pelo CPU, na forma física desejada para os operar. Esta pode ser um caracter (letra, figura, etc.), um ecrã, luzes acendendo no sentido ascendente ou descendente, manipulações do sistema exterior, dependendo do valor lido do exterior.

Nota:
As unidades de entrada e de saída dos sistemas de microcomputadores são seleccionados de acordo com as características requeridas pelas grandezas físicas exteriores. Todavia, as unidades *standard* de entrada e de saída disponíveis são capazes de dar resposta às funções desejadas.

A maior parte dos microcomputadores são operados por meio de um teclado, o qual traduz uma tecla premida num valor numérico. Uma unidade de entrada com a função de um teclado traduz uma tensão eléctrica num valor numérico. De modo análogo, existem unidades de entrada que traduzem comutadores de accionamento para grandezas numéricas.

1.3. UNIDADE DE MEMÓRIA

A memória é acedida frequentemente pelo CPU para o processamento dos dados.

A unidade de memória é composta por um conjunto de células; o tamanho da memória depende do número de células que a compõem; cada célula permite armazenar um valor binário de 8 dígitos que é designado de "*byte*". O valor armazenado em cada célula é chamado de "dados" da célula.

O tamanho dos dados e da memória é limitado e depende do microprocessador utilizado. A dimensão dos dados é definido pelo tamanho da célula; a dimensão da memória é definida pelo tipo de CPU.

Os microprocessadores 6800 da Motorola, 8085 da Intel, Z80 da Zilog e 6502 da MOS Technology possuem 8 bits de informação para os dados. Os dados processados e descodificados por estes CPUs são números binários de 8 bits, capazes de representar os números inteiros no intervalo [0,.., 255].

Associado a cada célula da memória está um número de série chamado "endereço". Este permite que uma célula seja endereçada, possibilitando ler o valor guardado ou escrever um novo valor na memória.

A disposição das células de memória corresponde a um empilhamento de células adjacentes, em que o endereço decresce no sentido da base para o topo da pilha, como mostra a figura seguinte:

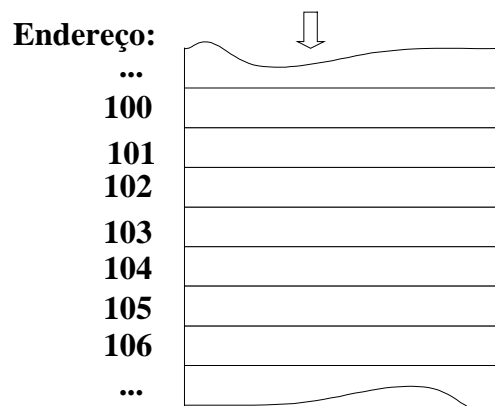



Figura 1.2 – **Pilha de células da memória**

Os microprocessadores indicados possuem um endereço de 16 bits de tamanho, permitindo endereçar até 65.536 células de memória diferentes. A um valor de 16 bits chama-se "word".

O CPU acede à memória e às outras unidades através de ligações chamadas de "barramentos" (em inglês Bus). Existem três tipos diferentes de *Bus*:

- Address Bus: barramento de endereços;

- Data Bus: barramento de dados;
- Control Bus: barramento de controlo.

Operação de leitura dos dados:

- Quando o CPU pretende ler os dados guardados na célula de endereço 354, por exemplo, o *Address Bus* é colocado igual a 354;
- O *Control Bus* habilita a memória a disponibilizar os dados (*data*) que lhe são solicitados;
- Os dados guardados na célula de memória de endereço 354 são então transferidos para o *Data Bus* que os conduzirá até ao CPU.

No entanto, embora o CPU receba o conteúdo da célula, os dados guardados não são alterados.

Operação de escrita dos dados:

- Quando o CPU pretende armazenar os dados numa célula de memória de endereço 356, por exemplo, o *Address Bus* é colocado a 356;
- O CPU acciona o *Control Bus* para habilitar a ordem de escrita dos dados na memória;
- Os dados são transferidos do CPU para a célula de memória de endereço 356 através do *Data Bus*.

Estes dados irão substituir o conteúdo armazenado anteriormente na célula de memória.

Os dados armazenados na memória representam duas grandezas:

- Valores processados ou a processar pelo CPU;
- Instruções que constituem o programa a ser executado pelo CPU.

O CPU traduz os dados em instruções, as quais correspondem a acções a ser realizadas.

1.4. IMPULSOS DO RELÓGIO

As operações realizadas pelo CPU são executadas passo-a-passo, comandadas pelos "impulsos do relógio" (em inglês Clock).

Exemplo:

- O CPU endereça uma célula de memória, a qual contém o código da instrução a ser executada;
- De seguida o CPU lê, descodifica e processa a instrução.

O mesmo procedimento é repetido para cada linha de código do programa. O tempo associado a cada procedimento é definido pelo *Clock* externo, o qual regula a duração em que as operações são processadas.

É muito importante garantir que a velocidade de execução das linhas de um programa esteja sincronizada com os tempos de acesso relativos às componentes acedidas e controladas pelo CPU.

1.5. UNIDADE DE PROCESSAMENTO CENTRAL

A compreensão do princípio de funcionamento dos microprocessadores requer que se conheça a sua constituição interna e as suas funções básicas. O CPU é composto pelas seguintes partes:

- Control Unit (CU): unidade de controlo;
- Arithmetic and Logic Unit (ALU): unidade aritmética e lógica;
- Register (R): registo.

Na figura seguinte estão representados os elementos principais que compõem o CPU, assim como os barramentos de ligação com as outras unidades:

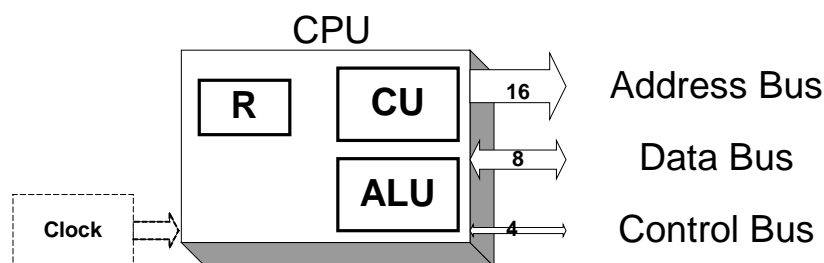



Figura 1.3 – **Estrutura básica do CPU**

1.5.1. UNIDADE DE CONTROLO

A unidade de controlo (CU) é a mais complexa do CPU. O CU recebe o código da instrução a ser executada, descodifica-o e processa os passos correspondentes à instrução.

1.5.2. UNIDADE ARITMÉTICA E LÓGICA

A unidade aritmética e lógica (ALU) é responsável pela execução das operações lógicas e aritméticas do CPU, de acordo com a instrução lida e descodificada.

A ALU recebe dois tipos de dados:

- Os provenientes de dois registos;
- Os que vêm de um registo e de uma célula de memória.

1.5.3. REGISTO

O registo é uma unidade de memória capaz de conter apenas um número. Esse número representa um dado ou um endereço, dependendo da função do registo; o tamanho dos registos pode ser de 8 ou de 16 bits, depende do seu tipo.

O registo do CPU mais utilizado é o Acumulador, designado por registo A. Grande parte dos dados fluem no CPU através deste registo.

O registo Program Counter (PC) de 16 bits é também importante, visto que guarda o endereço da instrução a ser processada. Assim que o dado da instrução é transferido da memória para o CPU, o PC é incrementado automaticamente uma unidade, passando a indicar o endereço da célula de memória seguinte.

2. CARACTERÍSTICAS DO MICROCOMPUTADOR

O CPU está ligado às unidades que compõem o microcomputador através dos barramentos (*bus*). A Figura 2.1 exemplifica a estrutura destas ligações.

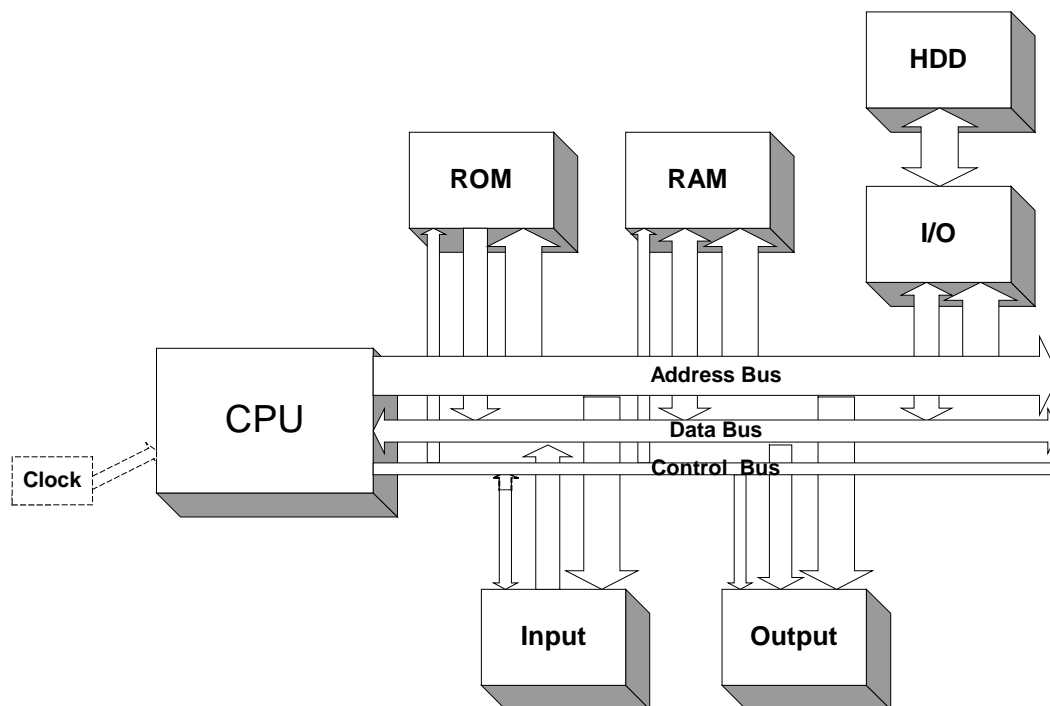



Figura 2.1 – **Estrutura do microcomputador**

2.1. BARRAMENTOS

Os *bus* consistem num conjunto de linhas que ligam as unidades do microcomputador ao CPU. Cada linha tem associado um estado lógico, “0” ou “1”.

O *Address Bus* do 8085 é composto por 16 linhas, as quais podem gerar 65.536 (ou 64k) endereços diferentes. Numa memória nunca existem duas células com o mesmo endereço.

O *Data Bus* do 8085 corresponde a um conjunto de 8 linhas, através das quais fluem números binários entre o CPU e as outras unidades.

Quando um dado de 16 bits é guardado são ocupadas duas células de memória adjacentes. Ao ler uma *word* o CPU necessita do endereço de duas células; o seu valor é transferido em duas etapas, um *byte* de cada vez.

O Control Bus do 8085 é composto por várias linhas, tendo cada uma delas uma função específica. As três principais linhas de controlo são as seguintes:

- RD': linha de leitura (ReaD) activa a zero;
- WR': linha de escrita (WRite) activa a zero;
- IO/M': linha de entrada (Input)/saída (Output) da memória (Memory) activa a zero.

Operação de leitura de dados:

- Quando o CPU efectua uma leitura de dados guardados na memória, o endereço da célula é colocado no *Address Bus*;
- As linhas IO/M' e RD' do *Control Bus* são colocadas a "0" (activação negativa), definindo a operação de acesso à memória para leitura de dados;
- Os dados (*data*) são colocados no *Data Bus* para serem enviados ao CPU.

Operação de escrita de dados:

- Quando o CPU pretende escrever os dados na memória, coloca no *Address Bus* o endereço da célula de memória que recebe o valor;
- As linhas IO/M' e WR' do *Control Bus* são activadas a "0" para a operação de acesso à memória e escrita de dados;
- Os dados são colocados no *Data Bus*;
- Depois, a célula endereçada guarda os novos dados, apagando os anteriores.

Quando o CPU comunica com uma unidade de entrada/saída (I/O, Input/Output), a linha IO/M' do *Control Bus* é colocada a "1".

Cada unidade ligada ao CPU possui um descodificador. A sua função é descodificar o endereço do *Address Bus* e identificar o estado das linhas IO/M', WR' e RD'.

Quando o descodificador da unidade de memória identifica que a linha WR' está no estado "0" então a informação presente no *Data Bus* deve ser escrita na célula de memória cujo endereço está presente no *Address Bus*.

Se a linha RD' está no estado "0" então o conteúdo da célula de memória de endereço identificado no *Address Bus* deve ser lido e colocado no *Data Bus*.

2.2. MEMÓRIAS

Existem três tipos de memórias:

1ª Random Access Memory (RAM): memória de escrita/leitura de acesso aleatório;

2ª Read Only Memory (ROM): memória exclusivamente de leitura;

3ª Hard Disk Device (HDD): memória magnética de escrita/leitura.

A memória RAM permite ao CPU armazenar dados provenientes das unidade de I/O, assim como dados resultantes do processamento do programa. O utilizador pode escrever ou alterar vários programas nesta memória.

Quando a fonte de alimentação do microcomputador é desligada, todos os dados armazenados na memória RAM são apagados. O conteúdo desta memória é constituído por valores aleatórios no momento em que a fonte é ligada.

A memória ROM guarda os dados e as instruções de forma permanente. A escrita de informação nesta memória é previamente feita por um sistema externo, antes da memória ser instalada no microcomputador. Por isso, o CPU apenas pode ler a informação presente na ROM, não a pode escrever.

A vantagem da memória ROM é que os dados guardados não se apagam quando a fonte de alimentação é desligada. Daí que a ROM seja utilizada para guardar os programas básicos para o funcionamento do microcomputador. Estes programas são designados de programa monitor por serem o sistema operativo do microcomputador.

As memórias magnéticas comunicam com o microcomputador através dos barramentos mas necessitam de ser ligadas às unidades de *Interface*.

As memórias magnéticas permitem a escrita e a leitura de dados; têm a forma de discos (*hard disk*), disquetes (*floppy disk*), cassetes, etc.; permitem o armazenamento dos dados e dos programas de forma não volátil; têm uma capacidade muito superior às outras memórias; têm um tamanho bastante maior do que as restantes memórias.

As memórias magnéticas têm a desvantagem de necessitarem de *interfaces* para poderem ser utilizadas e têm um tempo de acesso aos dados, quer para a leitura quer para a escrita, muito maior do que as memórias ROM ou RAM.

2.3. FUNCIONAMENTO DO MICROPROCESSADOR

Baseia-se em três princípios o funcionamento do microprocessador:

1º. Todas as unidades ligadas ao CPU estão conectadas em paralelo, através dos barramentos de dados, de endereços e de controlo;

2º. Qualquer unidade não endereçada pelo CPU está desligada do *Data Bus* e duas unidades não podem ter o mesmo endereço;

3º. O CPU funciona de forma sincronizada, em que o *clock* permite a sincronia da execução das operações através de etapas (ou passos).

O CPU funciona de acordo com o programa que está a ser executado, guardado na memória em células adjacentes. Este programa corresponde a um conjunto de números binários que representam as instruções e os dados.

Na fase de arranque do CPU 8085, o endereço da célula de memória inicial, definido por defeito, é 0000h. A parte da memória correspondente a este endereço é chamada de memória ROM. Este valor, expresso em hexadecimal e que corresponde a um número de 16 bits, endereça a primeira célula de memória e é carregado no PC (*program counter*). O CPU lê o registo PC para poder endereçar a instrução a ser executada na unidade de controlo (CU).

A unidade de controlo descodifica e processa a instrução, numa operação designada de Instruction Fetch – fase *fetch* da instrução. Após cada instrução executada, o PC é incrementado de uma unidade, passando a indicar o endereço da célula de memória que contém o código da instrução seguinte.

A célula de endereço 0000h contém o código da instrução que envia o CPU para o endereço da primeira célula do programa principal, também chamado de programa monitor.

A unidade de controlo utiliza igualmente dados de 2 e 3 *bytes* em grande parte das suas instruções. Assim, algumas instruções ocupam 2 e 3 células de memória adjacentes. O CPU identifica o tamanho da instrução, executando os sucessivos dados que a compõem.

A unidade de controlo descodifica e executa cada uma das instruções, sendo o PC (*program counter*) automaticamente incrementado de uma unidade, ficando a apontar para a próxima instrução.

Cada instrução é executada em duas etapas:
1ª Fetch: fase de preparação;
2ª Execute: fase de execução.

Vejamos o processamento de instruções típicas pelo CPU:
Exemplo 1: transferência do valor de um registo para uma célula de memória – a instrução inclui uma referência à memória e ao seu endereço.

Exemplo 2: transferência de dados entre registos – a instrução que efectua esta transferência é executada dentro do CPU.

Exemplo 3: instruções de funções aritméticas e lógicas (ALU) – a unidade de controlo é apoiada pela ALU para o processamento da instrução.

3. MODOS DE ENDEREÇAMENTO

O CPU necessita de dados para processar as instruções. Existem vários métodos de manipulação de dados utilizados pelo CPU, os quais são designados de modos de endereçamento.

O código da instrução, da designação inglesa Operation Code (opcode), tem inerente o modo de endereçamento processado pelo CPU.

3.1. MODO DE ENDEREÇAMENTO IMPLÍCITO

Neste modo de endereçamento está implícito a função da instrução. No endereçamento implícito não há campo variáveis após o *opcode*.

Tabela 3.1 – **Exemplos de modo de endereçamento implícito**

Opcode	*Mnemónica*	*Comentários*
37	STC	- Activação da *flag* de *carry*
3F	CMC	- Complemento da *flag* de *carry*

3.2. MODO DE ENDEREÇAMENTO IMEDIATO

Neste modo de endereçamento está indicado o valor do operando a seguir ao *opcode*.

Tabela 3.2 – **Exemplo de modo de endereçamento imediato**

Opcode	*Mnemónica*	*Comentário*
3E	MVI A, 55	- Guarda o valor 55H no registo A
55		

As instruções com endereçamento imediato têm dois *bytes* de tamanho: um para o *opcode* e outro para o operando.

Mas existe ainda uma instrução de três *bytes*, a única. Neste caso, o valor indicado tem dois *bytes* de tamanho, sendo utilizados dois registos para o armazenamento do dado.

Tabela 3.3 – **Exemplos de modo de endereçamento imediato de 3 *bytes***

Opcode	*Mnemónica*	*Comentário*
01	LXI B, 4010	- O valor 4010H é carregado no par de registos B-E
10		
40		

Após executada a instrução "LXI B, 4010", o registo B guarda o valor 40H e o registo E guarda o valor 10H.

3.3. MODO DE ENDEREÇAMENTO POR REGISTO

Neste modo de endereçamento são endereçados registos mas não a memória. O *opcode* para este endereçamento especifica os registo utilizados pela CPU no seu processamento.

Tabela 3.4 – **Exemplos de modo de endereçamento por registo**

Opcode	*Mnemónica*	*Comentário*
89	MOV A, B	- Transferência do conteúdo do registo B para o registo A (Acumulador)
3C	INR A	- Incremento de uma unidade do registo A

As instruções que utilizam o modo de endereçamento por registo têm apenas um *byte* de tamanho.

3.4. MODO DE ENDEREÇAMENTO DIRECTO

Neste modo de endereçamento é especificado o endereço da célula de memória que contém os dados a utilizar, após o *opcode*. As instruções que utilizam o modo de endereçamento directo têm três *bytes* de tamanho: um para o *opcode* e dois para o endereço (*Address High–Address Low*).

Tabela 3.5 – **Exemplo de modo de endereçamento directo**

Opcode	*Mnemónica*	*Comentário*
3A	LDA 2AA1	- Transferência do conteúdo da célula de
A1		memória 2AA1 para o registo A (Acumulador)
2A		

Após o processamento da instrução o Acumulador guarda o dado da célula de endereço 2AA1.

3.5. MODO DE ENDEREÇAMENTO INDIRECTO

O modo de endereçamento indirecto requer a utilização de dois registos, os quais funcionam como apontadores da célula de memória. Para isso é destinado o par de registos H-L que guarda o endereço de 16 *bits* da célula a aceder pelo CPU.

O registo H guarda o *byte* mais significativo do endereço (*Address* ***H****igh*), enquanto que o registo L guarda o *byte* menos significativo do endereço (*Address* ***L****ow*).

Este modo de endereçamento requer que os registos H-L sejam previamente carregados com o endereço da célula que se deseja apontar, antes de ser processada a instrução.

Tabela 3.6 – **Exemplo de modo de endereçamento indirecto**

Opcode	*Mnemónica*	*Comentário*
7E	MOV A, M[1]	- Transferência do conteúdo da célula de memória "M" de endereço H-L para o registo A

[1]A letra "M" representa a célula de memória apontada pelo par de registos H-L.

Após o processamento da instrução, o Acumulador guarda o dado da célula de endereço H-L.

Quando se pretende aceder a um conjunto de células adjacentes para operações com *Strings* ou *Arrays*, deve-se utilizar o modo de endereçamento indirecto. Endereçar um bloco de células adjacentes de memória torna-se fácil com este endereçamento, visto que basta incrementar ou decrementar o registo que funciona como apontador para aceder à célula seguinte.

3.6. CAMPOS VARIÁVEIS

Um programa armazenado na memória de um computador pode conter variáveis no seu código, declaradas no início ou atribuídas ao longo da listagem de instruções. Durante a execução do programa pelo CPU, o valor dessas variáveis vai sendo especificado.

Este tipo de programa permite a realização de múltiplas aplicações pois não define os dados nas instruções mas nos endereços das células de memória que os contêm. Os dados podem ser também definidos nos endereços das células de memória onde serão escritos.

Ao utilizar campos variáveis, o conteúdo de uma célula de memória pode mudar sem que a estrutura de funcionamento do programa se altere.

Quando se escreve um programa com uma estrutura variável, uma parte da memória RAM deve ser reservada para as células que irão conter os dados relativos às variáveis. Os endereços correspondentes a estas células devem ficar fora do intervalo de endereços utilizados para guardar o programa e nunca dentro do mesmo.

As instruções que manipulam variáveis requerem a indicação do endereço onde o seu valor concreto se encontra para poderem ser processadas.

4. FLAGS

4.1. INTRODUÇÃO

As *flags* são células de um único *bit*, existentes no CPU num registo designado de Program Status Word (PSW). Cada *flag* tem um nome próprio e opera independentemente.

As *flags* são utilizadas nas instruções de salto condicional, visto que a condição para o "jump" depende do estado de uma *flag*. Como o PSW possui várias *flags*, também haverá várias instruções de salto condicional.

Uma instrução de salto condicional corresponde a uma operação do tipo "Se A > B então ir para ...". Numa linguagem de alto nível esta operação pode ser implementada do seguinte modo:

IF A > B GOTO...

Ao nível do CPU, a instrução de salto condicional pode ser processada em duas fases:

1ª O ALU realiza a comparação de A com B. Esta operação vai provocar a alteração do estado das *flags*.

2ª A instrução de salto condicional é processada no ALU. De seguida é transmitido ao CU o endereço da instrução seguinte.

O microprocessador 8085 possui cinco diferentes *flags*:

1. *Flag* de *Carry* (CY);
2. *Flag* de *Auxiliary Carry* (AC);
3. *Flag* de *Zero* (Z);
4. *Flag* de *Signal* (S);
5. *Flag* de *Parity* (P).

4.2. TIPOS DE FLAGS

4.2.1. FLAG DE CARRY

Esta *flag* reflecte o *carry* da última operação efectuada pelo ALU. A *flag* de *Carry* (C) é activada (estado lógico "1") quando um *bit* a "1" transborda da 8ª para a 9ª coluna em resultado da operação processada pelo ALU.

Se a adição dos dois números binários de oito *bits* resulta num número de 9 *bits*, o nono *bit*, resultante do transbordo, é guardado na *flag* de *carry*.

Exemplo:

```
   10110011
+  01110010
[1]00100101
```

4.2.2. FLAG DE AUX CARRY

Esta *flag* indica se houve *carry* da 4ª (*bit* 3) para a 5ª coluna (*bit* 4), na última operação realizada pelo ALU. Se houve, a *flag Aux Carry* (AC) é activada (estado lógico "1").

A *flag Aux Carry* (AC) é utilizada em operações com números em código BCD ou em operações com grupos de 4 bits.

Exemplo:

```
   1000  1101
+  0110  0110
   111[1]0011
```

O *bit* da 5ª coluna foi adicionado ao *bit* a "1" vindo da 4ª coluna.

4.2.3. FLAG DE ZERO

Esta *flag* é activada quando o resultado da operação efectuada pelo ALU é zero. A *flag* de *Zero* (Z) activa corresponde a Z=1.

Exemplo:

```
   01101101
–  01101101
   00000000
```

A *flag* Z é activada visto que o resultado da subtracção é zero.

4.2.4. FLAG DE SIGNAL

Esta *flag* indica se o resultado da operação executada pelo ALU é positivo ou negativo. A *flag* de *Signal* (S) corresponde ao MSB, ou *bit* da 8ª coluna. Se este *bit* for "1" o resultado da operação é negativo, senão é positivo.

Exemplo:

```
   00110010
–  01110011
[1]10111111
```

A *flag* S é activada porque o resultado da subtracção é negativo, MSB=1.

4.2.5. FLAG DE PARITY

Esta *flag* dá a informação quanto ao número de "1"s existentes no resultado da última operação efectuada pelo ALU. A *flag* de *Parity* (P) é activada quando o número de *bits* a "1" é par, ou seja, paridade par de "1"s.

A *flag* de *Parity* é utilizada na comunicação entre computadores para o controlo de erros da transmissão.

Exemplo:

```
   10010011
+  00110010
   11000101
```

O resultado desta operação tem quatro "1"s, logo a *flag* P toma o valor "1".

5. ESTRUTURA BÁSICA DO SOFTWARE

5.1. INTRODUÇÃO

Um sistema computacional qualquer é composto por duas distintas partes:

1ª Hardware – corresponde à parte física do sistema;

2ª software – corresponde à parte não física do sistema, como seja a informação digital guardada ou processada pelo sistema.

Nos primórdios do desenvolvimento dos microcomputadores, o *hardware* correspondia ao maior esforço de projecto e de realização do sistema. Actualmente, os sistemas computacionais usam componentes *standard*, produzidos em série e em grandes quantidades.

Por outro lado, o *software* a ser processado pelo sistema computacional foi tornando-se cada vez mais complexo e elaborado. Nos dias de hoje a concepção e o projecto de aplicações de *software* torna-se mais relevante e envolve maiores custos do que o *hardware*.

O CPU lê os dados da unidade de memória ou da unidade de entrada e, depois de processados segundo o programa armazenado em memória, envia-os para a unidade de saída ou escreve-os na memória.

Uma aplicação de *software* corresponde a um conjunto de instruções que o CPU processa. A aplicação é escrita pelo programador. O CPU interage com o sistema enquanto processa os dados. Contudo, o CPU não tem a capacidade de tomar decisões autónomas, está dependente das instruções que executa.

Qualquer sistema de microcomputador tem uma unidade de entrada e uma unidade de saída. A ligação a essas unidade externas é feita através de portos que podem funcionar como entradas e saídas. O programa que deseja aceder às unidades externas tem instruções com funções específicas para esse efeito.

Um microcomputador pode substituir qualquer sistema digital, assim como um programa pode executar uma função complexa, decompondo-a em funções mais simples e definidas para processamento.

A definição da funcionalidade do sistema computacional por meio de *software* resulta em várias vantagens para o seu emprego, tais como:

1ª A execução do *software* pode ser feita em hardware standard, por isso mais económico;

2ª É possível modificar o funcionamento do sistema computacional através de simples alterações no *software*;

3ª Um sistema de processamento pode executar um grande número de funções implementadas em *software*.

5.2. REALIZAÇÃO DE SOFTWARE

Para que um programa possa ser realizado devem ser executadas as seguintes etapas:

1ª Definição do problema;

2ª Definição do algoritmo;

3ª Representação do algoritmo em *flowchart*;

4ª Escrita do programa;

5ª Execução e teste do programa.

A realização de um programa não obriga a que sejam necessariamente estas as etapas para a sua realização, podem ser outras. Cada programador terá o seu métodos e os seus segredos.

Todavia, evitam-se grandes dificuldades e obtêm-se melhores resultados se se seguir naturalmente as etapas atrás definidas. Sobretudo, é importante que a primeira etapa, a definição do problema, seja bem estruturada. Muitos contratempos advêm de não ter sido compreendido como deve ser o problema proposto.

5.2.1. DEFINIÇÃO DO PROBLEMA

Neste estágio é listadas toda a informação relativa ao problema e aos requisitos do microcomputador onde irá ser implementado. São igualmente listados os dados lidos das entradas e das saídas pretendidas para o computador.

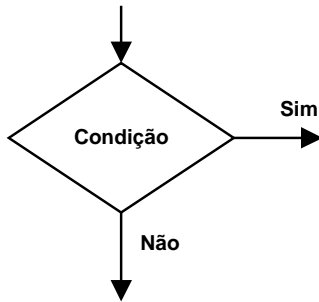
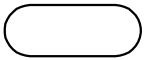
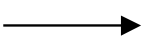
Nesta etapa os detalhes da fase de execução do programa não são importantes. Pelo contrário, espera-se que possa haver alterações no programa, quer na entrada quer na saída de dados.

O programador deverá nesta fase estar ciente das limitações do sistema. Isto vai permitir-lhe encontrar a melhor solução para o problema.

5.2.2. DEFINIÇÃO DO ALGORITMO

Um algoritmo é uma especificação dos estágios de execução de um determinado problema. O algoritmo permite definir os passos a seguir e, deste modo, encontrar a sequência das etapas na resolução do problema.

Um algoritmo é escrito em linguagem natural e utiliza termos gerais.

5.2.3. REPRESENTAÇÃO DO ALGORITMO EM FLOWCHART

A representação em *flowchart* do algoritmo corresponde a uma descrição gráfica das etapas a serem processadas.

Cada etapa do *flowchart* é escrita através de símbolos; transita-se de uma para outra etapa por meio da indicação de uma seta. As especificações mais detalhadas e minuciosas correspondem às instruções do programa, tendo em vista facilitar a escrita do seu código.

A representação em *flowchart* é um meio destinado a simplificar e a facilitar a compreensão do problema, ajudando a entender as condições e a forma como o programa é executado.

A utilização da representação em *flowchart* permite efectuar alterações e perceber, por simulação, o que o programa realmente faz. Os símbolos que compõem a representação em *flowchart* permitem representar os vários tipos de instruções ao longo das etapas do programa.

Tabela 5.1 – **Símbolos utilizados na representação em *flowchart***

Símbolo	*Descrição*
	- Entrada /saída de dados
Condição; Sim; Não	- Salto condicional – decisão binária (If – Then – Else)
	- Processo, operação
	- Início / Fim
	- Ligação entre blocos
	- Linha de fluxo

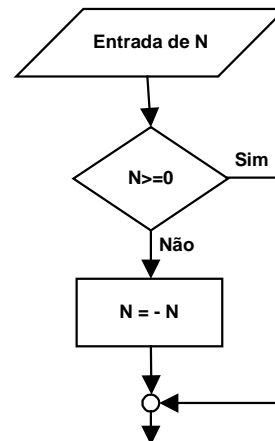
Exemplo de aplicação: programa para calcular o valor absoluto de um número inteiro qualquer.

Algoritmo:

1. Recebe o número (n);
2. Testa se o n é positivo;

3.1. Se o n é positivo não é alterado;

3.2. Se o n é negativo converte-o em positivo.

Representação em *Flowchart*:

A execução de uma instrução deve estar ligada ao destino que recebe o dado processado, através de uma linha de fluxo.

Uma representação gráfica, do género do *flowchart*, ajuda a implementar os algoritmos, ao mesmo tempo que facilita a compreensão dos programas.

5.2.4. ESCRITA DO PROGRAMA

O processamento computorizado de um programa permite executar repetidamente um conjunto de instruções com facilidade.

A pessoa que concebe e realiza programas é chamada de "programador". Para explorar as potencialidades do microcomputador, o programador deve estar familiarizado com a linguagem e as instruções a utilizar nos programas que implementa.

Um programa deve ser escrito numa linguagem interpretável pelos microprocessadores. O CPU executa apenas as instruções em binário; a memória armazena apenas a informação binária. Contudo, seria extremamente difícil escrever programas em linguagem binária.

O processo de tradução mais fácil é utilizar um sistema de codificação das instruções e dados em código hexadecimal. Neste sistema é convertida a informação hexadecimal introduzida em código binário na unidade de entrada.

5.2.5. EXECUÇÃO E TESTE DO PROGRAMA

A verificação de um programa é necessária para determinar se a sua execução está correcta. Um programa é testado depois de escrito.

O teste a um programa consiste na sua execução, assumindo valores particulares para as varáveis de entrada do programa. Ao erro detectado durante a execução de um programa chama-se Bug. A aplicação utilizada para detectar e corrigir *bugs* chama-se Debugger.

O teste de um programa tem em vista detectar erros de programação ou verificar a sua funcionalidade. Ao fazer o *debugging* do sistema é possível detectar erros quer de *hardware* – enganos de ligações ou de avarias dos circuitos – quer de *software* – enganos no programa –.

O teste ao funcionamento do programa pode ser feito através de duas operações distintas:

- Single step – o CPU para a sua actividade em qualquer ciclo, podendo ser visualizados os endereços e os dados que ima ser processados;
- Instruction step – o CPU para a sua actividade no momento em que recolhe da memória o *opcode* da próxima instrução a ser executada.

Para retomar à sua actividade normal o CPU necessita de uma acção externa.

5.3. LINGUAGEM MÁQUINA

Um programa é escrito numa linguagem que pode ser traduzido para a linguagem Máquina. Designa-se por linguagem Máquina a linguagem que utiliza apenas números binários ou hexadecimais na escrita dos programas.

Ao escrever um programa em linguagem Máquina o programador necessita de ter presente as seguintes informações:

1ª As instruções do CPU e os seus códigos;

2ª O endereço de memória da primeira instrução.

O endereço da célula onde está ou será colocado um determinado dado é escrito a seguir ao código de instrução.

O CPU 8085 possui instruções de um, dois e três *bytes*. O 1º *byte* a ser introduzido na memória corresponde ao código da instrução, o 2º *byte* corresponde ao dado a processar ou o 2º e o 3º *bytes* correspondem ao endereço da memória onde irá ser lido o dado.

Para tornar a escrita e a leitura de um programa em linguagem Máquina mais fácil, existem os campos de endereços e de código, aos quais se devem seguir os campos *label*, mnemónica e comentários.

Tabela 5.2 – **Estruturação de um programa numa forma inteligível**

Address	*Opcode*	*Label*	*Mnemónica*	*Comentários*
2000	3E	Start:	MVI A, 0FH	; carrega o Acumulador com o valor 0FH
2001	0F			
2002	D3		OUT 00H	; transfere o conteúdo do Acumulador para o porto de saída, de endereço 00H, ligando os 4 LEDs mais à direita
2003	00			

A *label* é utilizada para atribuir um nome a uma dada linha do programa, tornando mais fácil a sua identificação.

A mnemónica representa a instrução, sendo constituída por três componentes:

1ª Operação – abreviatura da palavra que descreve a operação realizada pela instrução;

2ª Operandos – dados utilizados na instrução;

3ª Modo de endereçamento – corresponde ao tipo de movimento dos dados mas nem sempre está definida na instrução.

O comentário descreve a instrução e fornece informação sobre a sua natureza. A leitura e a compreensão do programa torna-se difícil ou, por vezes, impossível quando um programa não é comentado.

Na linguagem Máquina há uma relação muito próxima entre os endereços e as mnemónicas. Um programa deve ter sempre indicado o seu endereço inicial.

Além desta linguagem existe mais outras duas: a linguagem Assembly e a linguagem de Alto Nível.

5.4. LINGUAGEM ASSEMBLY

A linguagem mais próxima da linguagem Máquina é a linguagem Assembly. As instruções em *Assembly* têm uma forma semelhante à mnemónica.

A linguagem *Assembly* tem várias vantagens, relativamente à linguagem Máquina, tais como:

1ª Não necessita de especificar a posição de memória correspondente à instrução "jump";

2ª As alterações ao programa são mais fáceis, bastando colocar ou retirar as instruções;
3ª A escrita do programa na memória é mais simples;
4ª A leitura e a detecção de erros do programa é mais rápida.

Um programa escrito em linguagem *Assembly* necessita de ser traduzido em linguagem Máquina para ser executado. A conversão da linguagem *Assembly* em linguagem Máquina é realizada pelo Assembler – programa que faz a conversão das linguagens quando executado –.

5.5. LINGUAGEM DE ALTO NÍVEL

Uma linguagem de programação de Alto Nível permite escrever as instruções numa forma bastante distante da linguagem Máquina. Por isso, torna-se mais fácil ao programador escrever um programa em linguagem de Alto Nível.

Exemplos de linguagens de Alto Nível são o Pascal, o Basic, o Fortran, o C, etc. Todavia, todo o programa escrito numa linguagem de Alto Nível necessita de ser traduzido em linguagem Máquina para que possa ser executado pelo microprocessador.

O programa que tem a seu cargo a tarefa da conversão da linguagem de Alto Nível para a linguagem Máquina é chamado de Compilador. Contudo, este programa tem um elevado número de instruções, uma grande complexidade e um tamanho considerável.

A relação entre as três linguagens, relativamente ao *hardware* e ao programador, está sintetizada na figura 5.1.

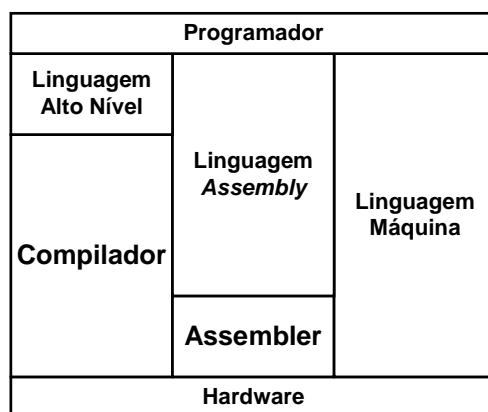



Figura 5.1 – **Relação entre as linguagens de programação**

5.6. CICLOS

Para implementar um ciclo utilizam-se quatro fases:

1ª Inicialização – definição das variáveis do ciclo, dos valores inicial e final do ciclo, das células de memória a serem acedidas pelas variáveis pertencentes ao ciclo;

2ª Instruções – conjunto de instruções a serem executadas repetidamente durante a activação do ciclo;

3ª Execução e teste – execução das instruções do ciclo, fazendo a actualização das variáveis, teste da condição de repetição do ciclo, sendo que o ciclo termina assim que a condição definida deixa de ser satisfeita;

4ª Conclusão – resultados da execução do ciclo, analisados e guardados na memória.

5.7. SALTOS

Um microprocessador executa dois tipos de saltos: salto incondicional e salto condicional. Para cada um deles está definida uma instrução de salto.

5.7.1. SALTO INCONDICIONAL

Ao executar uma instrução de salto incondicional o CPU salta incondicionalmente para a instrução do programa indicada pelo endereço ou pela *label* especificados. Esta instrução altera a sequência do endereço da instrução a ser executada, modificando a estrutura do programa.

5.7.2. SALTO CONDICIONAL

A execução de uma instrução de salto condicional é feita em duas fases:

1ª Teste – é analisada e testada a condição de salto de endereço;

2ª Salto – o CPU salta para o endereço especificado se a condição se verificar, senão o CPU segue para o endereço da instrução imediata.

Esta instrução de salto condicional é do tipo "jump if...", "salta se...". O resultado da execução desta instrução depende da veracidade da condição: se for verdadeira o salto é realizado, se for falsa passa para a instrução seguinte, sem haver salto.

A instrução de salto condicional corresponde a uma decisão binária na representação em *flowchart*.

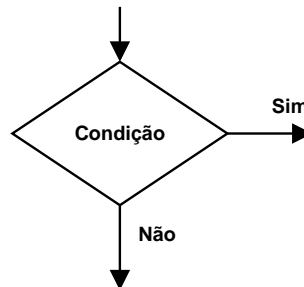



Figura 5.2 – **Símbolo da decisão binária**

O símbolo de *flowchart* de decisão binária representa correctamente a sequência do algoritmo para a instrução de salto condicional, facilitando a sua transposição para a linguagem *Assembly*.

A instrução de salto condicional permite realizar a execução repetida de um bloco de instruções. Chama-se Ciclo (loop) à repetição das instruções.

A representação em *flowchart* que incorpora um ciclo tem o seguinte aspecto:

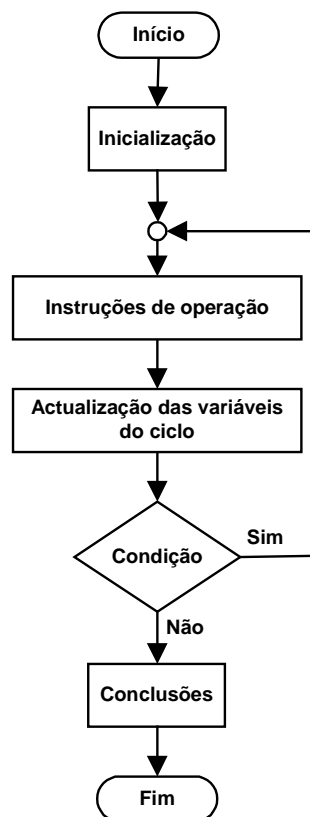



Figura 5.3 – ***Flowchart* do *loop***

Um programa pode ter mais do que um ciclo, geralmente tem vários e diferentes uns dos outros.

A instrução de salto condicional permite implementar ciclos cujo número de repetições não é conhecido à-priori, visto depender da condição especificada.

A instrução de salto condicional nem sempre realiza um *loop*; pode ser utilizada para seleccionar a execução de um bloco de instruções se uma dada condição for satisfeita ou outro bloco distinto se a condição não for satisfeita. A figura 5.4 corresponde a uma representação em *flowchart* desta situação.

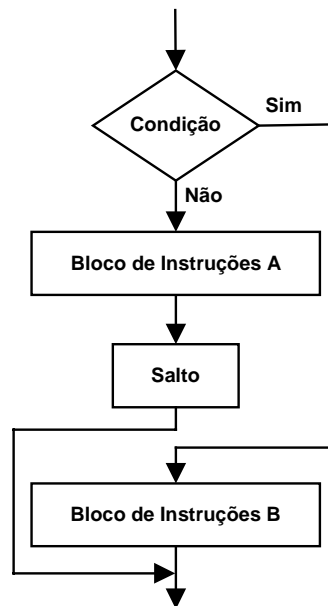



Figura 5.4 – **Representação em *Flowchart* da execução de dois blocos alternativos**

Um programa corresponde a um conjunto de instruções executadas sucessivamente, segundo uma determinada orientação.

A representação de um algoritmo em *flowchart* facilita a conversão da linguagem natural para a linguagem de programação definida.

5.8. TEMPORIZAÇÃO

Uma temporização consiste na execução de uma instrução que realiza uma contagem decrescente, desde um número definido até zero. A repetição da instrução ocupa o CPU

durante um certo intervalo de tempo, parecendo que o programa está parado. Chama-se loop de delay à temporização do CPU.

5.9. PROGRAMAÇÃO ESTRUTURADA

A representação em *flowchart* de um algoritmo, embora tendo diversas vantagens tem também as suas desvantagens. Se um programa tiver um grande número de condições e saltos condicionais, o *flowchart* será confuso e, por isso, de difícil leitura.

Um programa bem elaborado apresenta uma forma estruturada. Isso quer dizer que o programa está dividido em módulos, tendo cada um deles um determinado propósito.

Cada módulo é construído como se se tratasse de uma rotina, sendo chamado a partir do programa principal, de acordo com os requisitos pretendidos. Ler e analisar um programa estruturado por módulos torna-se relativamente simples.

5.9.1. ROTINA

O conjunto de instruções que implementam um *loop* de *delay* pode ser definido por uma entidade independente do programa principal designada por rotina.

Uma rotina é definida por duas instruções:

1ª A instrução de chamada à rotina (Call), colocada no programa principal e que provocará o salto incondicional do CPU para o endereço da sua primeira instrução;

2ª A instrução de retorno ao programa principal (Return), colocada no fim da rotina e que permite saltar incondicionalmente para o endereço da instrução imediata da instrução de chamada.

A rotina tem várias vantagens, tais como:

1ª Possibilita a modularidade do programa;

2ª Evita a repetição de um conjunto de instruções.

5.9.2. ETIQUETA

Para tornar a escrita de um programa que inclui *loops* e *jumps*, de leitura mais fácil utilizam-se etiquetas, geralmente designadas por labels. Uma *label* não tem significado para o CPU, serve apenas para facilitar a compreensão do funcionamento de um programa.

Uma *label* é um nome simples e expressivo, dado pelo programador, destinado a indicar um endereço ou uma zona do programa.

6. INSTRUÇÕES DO MICROPROCESSADOR 8085

6.1. INTRODUÇÃO

As instruções do microprocessador 8085 estão relacionadas com os registos e com as células de memória. O 8085 possui 80 instruções básicas. A maior parte delas resultam dos vários modos de endereçamento e dos diferentes registos.

Toda a instrução é identificada por um código (*opcode*) que depende da operação realizada, dos operandos envolvidos e do modo de endereçamento implícito. O *opcode* é um valor de dois dígitos, expresso em hexadecimal, de 8 bits de tamanho, utilizado para escrever um programa em linguagem máquina. O CPU 8085 tem 246 diferentes *opcodes*.

Para escrever um programa em linguagem *Assembly* utilizam-se as mnemónicas de instrução. A mnemónica representa a operação e os operandos usados na instrução.

A operação é expressa na mnemónica por 2 a 4 letras, geralmente resultantes do termo em inglês que a descreve.

Os operandos correspondem aos registos, à memória e aos valores envolvidos. Quando utilizados aos pares, os operandos são separados por uma vírgula. Os tipos de operandos utilizados são os seguintes:

1. Registos simples (A, B, C, D, E, H e L);
2. Registos duplos (BC, DE, HL, em que aparece na mnemónica a primeira letra), utilizados para dados de 16 bits;
3. Célula de memória (M), endereçada pelo par de registo HL, em modo de endereçamento indirecto;
4. Dados de 8 bits (byte) ou de 16 (dble);
5. Endereços de 16 bits (addr) para aceder às células de memória.

Cada instrução pode ter 1, 2 ou 3 *bytes* de tamanho: um *byte* para o *opcode* e 0, 1 ou 2 *bytes* para representar os operandos.

Para a execução de uma instrução o CPU necessita de vários ciclos do relógio (*clock cycle*). O número de *clock cycles* da CPU por instrução varia de 4 a 18, sendo uma característica própria de cada uma delas.

A execução de uma instrução pode afectar o estado de uma ou mais *flags*, visto reflectirem o estado do CPU.

6.2. CLASSIFICAÇÃO DAS INSTRUÇÕES

As instruções do microprocessador 8085 podem ser classificadas nos seguintes grupos:

1. Instrução de transferência de dados,
2. Instrução aritmética,
3. Instrução lógica,
4. Instrução de salto,
5. Instrução de uso da *stack*,
6. Instrução de entrada/saída (I/O),
7. Instrução de controlo.

6.3. INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

6.3.1. MOV r_d, r_s

Instrução que move (*MOVe*) o conteúdo do registo fonte (r_s) para o registo destino (r_d). Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e não afecta as *flags*.

Tabela 6.1 – ***Opcodes* de MOV r_d, r_s (em hexadecimal)**

r_d \ r_s	*A*	*B*	*C*	*D*	*E*	*H*	*L*
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	5F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6C	6D

6.3.2. MOV r_d, M

Instrução que move o conteúdo da célula de memória apontada pelo par de registos HL (em que o registo H guarda o *byte* de maior peso do endereço) para o registo destino (r_d). Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 7 ciclos do CPU para a sua execução e não afecta as *flags*.

Tabela 6.2 – ***Opcodes* de MOV r_d, M (em hexadecimal)**

M \ r_d	A	B	C	D	E	H	L
	7E	46	4E	56	5E	66	6E

6.3.3. MOV M, r_s,

Instrução que move o conteúdo o registo fonte (r_s) para a célula de memória apontada pelo par de registos HL. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 7 ciclos do CPU para a sua execução e não afecta as *flags*.

Tabela 6.3 – ***Opcodes* de MOV M, r_s (em hexadecimal)**

M \ *r_s*	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	77	70	71	72	73	74	75

6.3.4. MVI r_d, *byte*

Instrução que move de imediato (*MoVe Immediate*) o valor especificado de um *byte* para o registo r_d. Esta instrução tem dois *bytes* de tamanho, um para o *opcode* e outro para o dado, necessita de 7 ciclos do CPU para a sua execução e não afecta as *flags*.

Tabela 6.4 – ***Opcodes* de MVI r_d, *byte* (em hexadecimal)**

byte \ *r_d*	A	B	C	D	E	H	L
	3E	06	0E	16	1E	26	2E

6.3.5. MVI M, *byte*

Instrução que move de imediato o valor especificado de um *byte* para a célula de memória apontada pelo par de registos HL. Esta instrução tem dois *bytes* de tamanho, um para o *opcode* (de valor 36H) e outro para o dado, necessita de 10 ciclos do CPU para a sua execução e não afecta as *flags*.

6.3.6. LXI r_{dp}, *dble*

Instrução que carrega de imediato (*Load Immediate*) o valor especificado de dois *byte* (*double*) para o par de registos r_{dp}, em que aparece indicado o registo que guarda o *byte* de maior peso. Esta instrução tem três *bytes* de tamanho, um para o *opcode* e dois para o dado, necessita de 10 ciclos do CPU para a sua execução e não afecta as *flags*.

Tabela 6.5 – ***Opcodes* de LXI r_{dp}, *dble* (em hexadecimal)**

dble \ *r_{dp} par*	B	D	H	SP
	C	E	L	
	01	11	21	31

6.3.7. LDA addr

Instrução que carrega o Acumulador (*LoaD Accumulator*) directamente com o conteúdo da célula de memória cujo endereço é especificado. Esta instrução tem três *bytes* de tamanho, um para o *opcode* (de valor 3AH) e dois para o endereço, sendo o 2º *byte* guardo na memória os 8 bits de menor peso, necessita de 13 ciclos do CPU para a sua execução e não afecta as *flags*.

6.3.8. STA addr

Instrução que armazena o conteúdo do Acumulador (*STore Accumulator*) directamente na célula de memória de endereço especificado. Esta instrução tem três *bytes* de tamanho, um para o *opcode* (de valor 32H) e dois para o endereço, necessita de 13 ciclos do CPU para a sua execução e não afecta as *flags*.

6.3.9. LDAX r_{sp}

Instrução que carrega o Acumulador indirectamente com o conteúdo da célula de memória apontada pelo par de registos especificado. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 7 ciclos do CPU para a sua execução e não afecta as *flags*.

Tabela 6.6 – ***Opcodes* de LDAX r_{sp} (em hexadecimal)**

r_{dp} / par / (A)	B	D
	C	E
	0A	1A

6.3.10. STAX r_{dp}

Instrução que armazena o conteúdo do Acumulador indirectamente na célula de memória apontada pelo par de registos especificado. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 7 ciclos do CPU para a sua execução e não afecta as *flags*.

Tabela 6.7 – ***Opcodes* de STAX r_{dp} (em hexadecimal)**

r_{dp}	par	(A)
B	C	02
D	E	12

6.3.11. LHLD addr

Instrução que carrega os registo H e L directos (*Load H & L Direct*) com o conteúdo das duas células de memória adjacentes de endereço inicial especificado. O registo L guarda o conteúdo da célula de endereço addr e o registo H guarda o conteúdo da célula addr+1. Esta instrução tem três *bytes* de tamanho, um para o *opcode* (de valor 2AH) e dois para o endereço, necessita de 16 ciclos do CPU para a sua execução e não afecta as *flags*.

6.3.12. SHLD addr

Instrução que armazena o conteúdo dos registo H e L directos (*Store H & L Direct*) nas duas células de memória de endereço inicial especificado. Na célula de endereço addr será guardado o conteúdo do registo L e na célula de endereço addr+1 será guardado o conteúdo do registo H. Esta instrução tem três *bytes* de tamanho, um para o *opcode* (de valor 22H) e dois para o endereço, necessita de 16 ciclos do CPU para a sua execução e não afecta as *flags*.

6.3.13. XCHG

Instrução que troca (*eXCHanGe*) o conteúdo do par de registos HL pelo conteúdo do par de registos DE. O registo D recebe o conteúdo H e vice-versa. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor EBH), necessita de 4 ciclos do CPU para a sua execução e não afecta as *flags*.

6.4. INSTRUÇÕES ARITMÉTICAS

6.4.1. ADD r_S

Instrução que adiciona (*ADD*) o conteúdo do registo fonte (r_S) ao conteúdo do Acumulador que guarda o resultado. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar todas as *flags* (CY, AC, Z, P e S).

Tabela 6.8 – ***Opcodes* de ADD r_s (em hexadecimal)**

r_s / (A)	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	87	80	81	82	83	84	85

6.4.2. ADD M

Instrução que adiciona o conteúdo da célula de memória (M) apontada pelo par de registos HL ao conteúdo do Acumulador que guarda o resultado. Esta instrução tem um

byte de tamanho para o *opcode* (86H), necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.4.3. ADI *byte*

Instrução que adiciona de imediato (*ADd Immediate*) o valor especificado de um *byte* ao conteúdo do Acumulador que guarda o resultado. Esta instrução tem dois *byte* de tamanho, um para o *opcode* (de valor C6H) e outro para o dado, necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.4.4. ADC r_s

Instrução que adiciona com a *flag* de *Carry* (*Add with Carry*) o conteúdo do registo especificado ao conteúdo do Acumulador que guarda o resultado da soma dos três termos. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

Tabela 6.9 – ***Opcodes* de ADC r_s (em hexadecimal)**

(A) \ r_s	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	8F	88	89	8A	8B	8C	8D

6.4.5. ADC M

Instrução que adiciona com a *flag* de *Carry* o conteúdo da célula de memória (M), apontada pelo par de registos HL, ao conteúdo do Acumulador que guarda o resultado da soma. Esta instrução tem um *byte* de tamanho para o *opcode* (8EH), necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.4.6. ACI *byte*

Instrução que adiciona com a *flag* de *Carry* de imediato (*Add with Carry Immediate*) o valor especificado de um *byte* ao conteúdo do Acumulador que guarda o resultado da soma. Esta instrução tem dois *byte* de tamanho, um para o *opcode* (de valor CEH) e outro para o dado, necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.4.7. SUB r_s

Instrução que subtrai (*SUBtract*) o conteúdo do registo especificado ao conteúdo do Acumulador que guarda o resultado da diferença. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

Tabela 6.10 – ***Opcodes* de SUB r_s (em hexadecimal)**

(A) \ r_s	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	97	90	91	92	93	94	95

6.4.8. SUB M

Instrução que subtrai o conteúdo da célula de memória (M), apontada pelo par de registos HL, ao conteúdo do Acumulador que guarda o resultado da diferença. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 96H), necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.4.9. SUI *byte*

Instrução que subtrai de imediato (*SUbtract Immediate*) o valor especificado de um *byte* ao conteúdo do Acumulador que guarda o resultado da diferença. Esta instrução tem dois *byte* de tamanho, um para o *opcode* (de valor D6H) e outro para o dado, necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.4.10. SBB r_s

Instrução que subtrai com a *flag* de *Carry* (*SuBtract with Borrow*) o conteúdo do registo especificado ao conteúdo do Acumulador que guarda o resultado da diferença. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

Tabela 6.11 – ***Opcodes* de SBB r_s (em hexadecimal)**

(A) \ r_s	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	9F	98	99	9A	9B	9C	9D

6.4.11. SBB M

Instrução que subtrai com a *flag* de *Carry* o conteúdo da célula de memória (M), apontada pelo par de registos HL, ao conteúdo do Acumulador que guarda o resultado da diferença. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 9EH), necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.4.12. SBI *byte*

Instrução que subtrai com a *flag* de *Carry* de imediato (Subtract *with Borrow Immediate*) o valor especificado de um *byte* ao conteúdo do Acumulador que guarda o

resultado da diferença. Esta instrução tem dois *byte* de tamanho, um para o *opcode* (de valor DEH) e outro para o dado, necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.4.13. INR r

Instrução que incrementa (*INcRement*) em uma unidade o conteúdo do registo especificado que guarda o resultado. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar as *flags* S, Z, P e AC.

Tabela 6.12 – ***Opcodes* de INR r (em hexadecimal)**

r	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	3E	06	0E	16	1E	26	2E

6.4.14. INR M

Instrução que incrementa em uma unidade o conteúdo da célula de memória (M), apontada pelo par de registos HL, que guarda o resultado. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 34H), necessita de 10 ciclos do CPU para a sua execução e pode afectar todas as *flags* S, Z, P e AC.

6.4.15. DCR r

Instrução que decrementa (DeCRement) em uma unidade o conteúdo do registo especificado que guarda o resultado. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar as *flags* S, Z, P e AC.

Tabela 6.13 – ***Opcodes* de DCR r (em hexadecimal)**

r	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	3D	05	0D	15	1D	25	2D

6.4.16. DCR M

Instrução que decrementa em uma unidade o conteúdo da célula de memória (M), apontada pelo par de registos HL, que guarda o resultado. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 35H), necessita de 10 ciclos do CPU para a sua execução e pode afectar todas as *flags* S, Z, P e AC.

6.4.17. INX r_p

Instrução que incrementa em uma unidade o conteúdo do par de registos especificado (*Increment register pair*) que guarda o resultado. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 6 ciclos do CPU para a sua execução e as *flags* não são afectadas.

Tabela 6.14 – ***Opcodes* de INX r_p (em hexadecimal)**

r_p par	*B* / *C*	*D* / *E*	*H* / *L*	*SP*
	03	13	23	33

6.4.18. DCX r_p

Instrução que decrementa em uma unidade o conteúdo do par de registos especificado (DeC*rement register pair*) que guarda o resultado. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 6 ciclos do CPU para a sua execução e as *flags* não são afectadas.

Tabela 6.15 – ***Opcodes* de DCX r_p (em hexadecimal)**

r_p par	*B* / *C*	*D* / *E*	*H* / *L*	*SP*
	0B	1B	2B	3B

6.4.19. DAD r_p

Instrução que adiciona o conteúdo do par de registos especificado ao conteúdo do par de registos HL que guarda o resultado. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 10 ciclos do CPU para a sua execução e pode afectar a *flag* de *Carry*.

Tabela 6.16 – ***Opcodes* de DAD r_p (em hexadecimal)**

r_p par / *(HL)*	*B* / *C*	*D* / *E*	*H* / *L*	*SP*
	09	19	29	39

6.4.20. DAA

Instrução que ajusta a decimal o conteúdo do Acumulador (*Decimal Adjust Accumulator*), correspondente à adição de dois operandos expressos em BCD com a operação ADD, de modo que o resultado seja ainda em BCD. Esta instrução tem um

byte de tamanho para o *opcode* (de valor 27H), necessita de 4 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.5. INSTRUÇÕES LÓGICAS

6.5.1. ANA r_s

Instrução que realiza o E lógico do conteúdo binário do Acumulador (*ANd Accumulator*) com o conteúdo binário do registo especificado, guardando o resultado no Acumulador. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

Tabela 6.17 – ***Opcodes* de ANA r_s (em hexadecimal)**

(A) \ r_s	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	A7	A0	A1	A2	A3	A4	A5

6.5.2. ANA M

Instrução que realiza o E lógico do conteúdo binário do Acumulador com o conteúdo binário da célula de memória (M), apontada pelo par de registos HL, guardando o resultado no Acumulador. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor A6H), necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.5.3. ANI *byte*

Instrução que realiza o E lógico de imediato do conteúdo binário do Acumulador (*ANd Immediate*) com o valor binário especificado de um *byte*, guardando o resultado no Acumulador. Esta instrução tem dois *byte* de tamanho, um para o *opcode* (de valor E6H) e outro para o dado, necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.5.4. ORA r_s

Instrução que realiza o OU lógico do conteúdo binário do Acumulador (OR *Accumulator*) com o conteúdo binário do registo especificado, guardando o resultado no Acumulador. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

Tabela 6.18 – ***Opcodes* de ORA r_s (em hexadecimal)**

(A) \ r_s	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	B7	B0	B1	B2	B3	B4	B5

6.5.5. ORA M

Instrução que realiza o OU lógico do conteúdo binário do Acumulador com o conteúdo binário da célula de memória (M), apontada pelo par de registos HL, guardando o resultado no Acumulador. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor B6H), necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.5.6. ORI *byte*

Instrução que realiza o OU lógico de imediato do conteúdo binário do Acumulador (*OR Immediate*) com o valor binário especificado de um *byte*, guardando o resultado no Acumulador. Esta instrução tem dois *byte* de tamanho, um para o *opcode* (de valor F6H) e outro para o dado, necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.5.7. XRA r_s

Instrução que realiza o OU-Exclusivo lógico do conteúdo binário do Acumulador (*XoR Accumulator with register*) com o conteúdo binário do registo especificado, guardando o resultado no Acumulador. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

Tabela 6.19 – ***Opcodes* de XRA r_s (em hexadecimal)**

(A) \ r_s	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	AF	A8	A9	AA	AB	AC	AD

6.5.8. XRA M

Instrução que realiza o OU-Exclusivo lógico do conteúdo binário do Acumulador com o conteúdo binário da célula de memória (M), apontada pelo par de registos HL, guardando o resultado no Acumulador. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor AEH), necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.5.9. XRI *byte*

Instrução que realiza o OU-Exclusivo lógico de imediato (*XoR Immediate)* do conteúdo binário do Acumulador com o valor binário especificado de um *byte*, guardando o resultado no Acumulador. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor EEH), necessita de 7 ciclos do CPU para a sua execução e pode afectar todas as *flags*.

6.5.10. CMP r

Instrução que compara o conteúdo binário do Acumulador com o conteúdo binário do registo especificado (*CoMPare accumulator with register*), não altera os seus conteúdos mas altera o estado das *flags* como se ao 1º registo subtrai-se o 2º registo. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 4 ciclos do CPU para a sua execução e pode afectar as *flags* S, Z, P e AC.

Tabela 6.20 – ***Opcodes* de CMP r (em hexadecimal)**

r / (F)	*A*	*B*	*C*	*D*	*E*	*H*	*L*
	BF	B8	B9	BA	BB	BC	BD

6.5.11. CMP M

Instrução que compara o conteúdo binário do Acumulador com o conteúdo binário da célula de memória (M), apontada pelo par de registos HL, não altera os conteúdos guardados mas altera o estado das *flags* como se ao 1º registo subtraí-se o 2º valor. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor BEH), necessita de 7 ciclos do CPU para a sua execução e pode afectar as *flags* S, Z, P e AC.

6.5.12. CPI *byte*

Instrução que compara de imediato (*ComPare Immediate*) o conteúdo binário do Acumulador com o valor binário especificado de um *byte*, altera o estado das *flags* como se ao 1º registo subtrai-se o 2º valor. Esta instrução tem dois *byte* de tamanho, um para o *opcode* (de valor F6H) e outro para o dado, necessita de 7 ciclos do CPU para a sua execução e pode afectar as *flags* S, Z, P e AC.

6.5.13. RLC

Instrução que roda para a esquerda com a *flag* de *Carry* (*Rotate Left with Carry*) o conteúdo binário do Acumulador, passando o *bit* MSB para a posição de LSB e para a *flag* de *Carry*. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 07H), necessita de 4 ciclos do CPU para a sua execução e só afecta a *flag* de *Carry*.

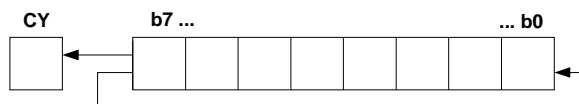



Figura 6.1 – **Rotação para a esquerda com a *flag* de *Carry* do conteúdo binário do Acumulador**

6.5.14. RRC

Instrução que roda para a direita com a *flag* de *Carry* (*Rotate Right with Carry*) o conteúdo binário do Acumulador, passando o *bit* LSB para a posição de MSB e para a *flag* de *Carry*. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 0FH), necessita de 4 ciclos do CPU para a sua execução e só afecta a *flag* de *Carry*.

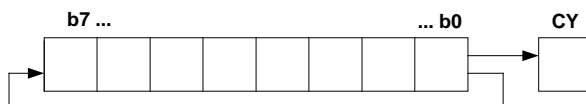



Figura 6.2 – **Rotação para a direita com a *flag* de *Carry* do conteúdo binário do Acumulador**

6.5.15. RAL

Instrução que roda o conteúdo binário do Acumulador para a esquerda, passando pela *flag* de *Carry* (*Rotate Accumulator Left through carry*), passando o *bit* MSB para a *flag* de *Carry* e esta para a posição de LSB. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 17H), necessita de 4 ciclos do CPU para a sua execução e só afecta a *flag* de *Carry*.

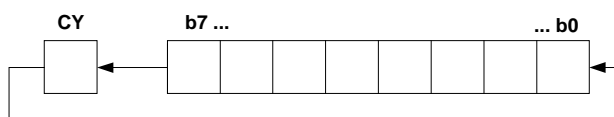



Figura 6.3 – **Rotação para a esquerda do conteúdo binário do Acumulador, passando pela *flag* de *Carry***

6.5.16. RAR

Instrução que roda o conteúdo binário do Acumulador para a direita, passando pela *flag* de *Carry* (*Rotate Accumulator Right through carry*), passando o *bit* LSB para a *flag* de *Carry* e esta para a posição de MSB. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 1FH), necessita de 4 ciclos do CPU para a sua execução e só afecta a *flag* de *Carry*.

b7 ... ... b0 CY

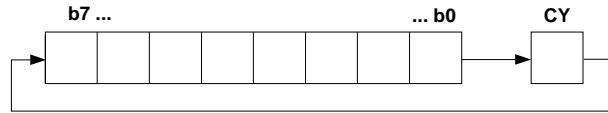



Figura 6.4 – **Rotação para a direita do conteúdo binário do Acumulador, passando pela *flag* de *Carry***

6.5.17. CMA

Instrução que complementa o conteúdo binário do Acumulador (*CoMplement Accumulator*), em complemento para um. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 2FH), necessita de 4 ciclos do CPU para a sua execução e as *flags* não são afectadas.

6.5.18. CMC

Instrução que complementa a *flag* de *Carry* (*CoMplement Carry*). Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 3FH), necessita de 4 ciclos do CPU para a sua execução e apenas a *flag* de *Carry* é afectada.

6.5.19. STC

Instrução que activa a *flag* de *Carry* (*SeT Carry*). Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 37H), necessita de 4 ciclos do CPU para a sua execução e apenas afecta a *flag Carry*.

6.6. INSTRUÇÕES DE SALTO

6.6.1. JMP addr

Instrução que provoca o salto (*JuMP*) incondicional do programa para o endereço especificado, alterando a sequência de execução das instruções. Esta instrução tem três *bytes* de tamanho, um para o *opcode* (de valor C3H) e dois para o endereço, necessita de 10 ciclos do CPU para a sua execução e não afecta as *flags*. Após executada, esta instrução coloca o contador do programa com o endereço especificado, PC = addr.

6.6.2. Jcondição addr

Instrução que provoca o salto condicional do programa para o endereço indicado, caso a condição especificada seja verdadeira (*Jump if "condição" is true*). Esta instrução tem

três *bytes* de tamanho, um para o *opcode* e dois para o endereço, necessita de 7 a 10 ciclos do CPU para a sua execução, não afecta as *flags e*, após executada, PC = addr.

Tabela 6.21 – **Instruções de salto condicional para várias condições**

Instrução	*Opcode*	*Condição*	*Observações*
JZ addr	CA	Z = 1	*Jump if Zero*
JNZ addr	C2	Z = 0	*Jump if Not Zero*
JC addr	DA	C = 1	*Jump if Carry*
JNC addr	D2	C = 0	*Jump if Not Carry*
JPE addr	EA	P = 1	*Jump if Parity Even* (par)
JPO addr	E2	P = 0	*Jump if Parity Odd* (ímpar)
JM addr	FA	S = 1	*Jump if Minus* (negativo)
JP addr	F2	S = 0	*Jump if Plus* (positivo)

6.6.3. PCHL

Instrução que provoca o salto incondicional indirecto do registo PC para o par de registos HL (*PC is moved to HL*). Esta instrução tem um *byte* de tamanho para o *opcode* (de valor E9H), necessita de 6 ciclos do CPU para a sua execução e não afecta as *flags*.

Após executada, esta instrução coloca o contador do programa com o conteúdo do par de registos HL, PC = HL.

6.6.4. CALL addr

Instrução de chamada (*CALL*) à rotina; guarda o endereço de retorno ao programa (correspondente à instrução imediatamente seguinte) no topo da pilha (*Stack*) e salta incondicionalmente para a instrução de endereço especificado. Esta instrução tem três *bytes* de tamanho, um para o *opcode* (de valor CDH) e dois para o endereço, necessita de 18 ciclos do CPU para a sua execução e não afecta as *flags*.

Após executada, a instrução CALL coloca o endereço de retorno na *Stack*, [SP-1] = *high* (PC_{return}), [SP-2] = *low* (PC_{return}), ficando SP = SP-2 e coloca o contador do programa com o endereço especificado, PC = addr.

6.6.5. CALLcondição addr

Instrução de chamada condicional à rotina para o endereço indicado se a condição especificada for verdadeira (*CALL if "condição" is true*); guarda o endereço de retorno no topo da *Stack* e salta incondicionalmente para a instrução do endereço indicado. Esta instrução tem três *bytes* de tamanho, um para o *opcode* e dois para o endereço, necessita de 9 a 18 ciclos do CPU para a sua execução e não afecta as *flags*.

Após a sua execução, a instrução CALLcondição coloca o endereço de retorno na *Stack*, [SP-1] = *high* (PC_{return}), [SP-2] = *low* (PC_{return}), fica SP = SP-2 e o contador do programa é colocado com o endereço especificado, PC = addr.

Tabela 6.22 – **Instruções de chamada condicional à rotina para várias condições**

Instrução	*Opcode*	*Condição*	*Observações*
CZ addr	CC	Z = 1	*Call if Zero*
CNZ addr	C4	Z = 0	*Call if Not Zero*
CC addr	DC	C = 1	*Call if Carry*
CNC addr	D4	C = 0	*Call if Not Carry*
CPE addr	EC	P = 1	*Call if Parity Even* (par)
CPO addr	E4	P = 0	*Call if Parity Odd* (ímpar)
CM addr	FC	S = 1	*Call if Minus* (negativo)
CP addr	F4	S = 0	*Call if Plus* (positivo)

6.6.6. RET

Instrução de retorno (*RETurn*) ao programa, após a execução da rotina; realiza o salto incondicional para o endereço de retorno ao programa, guardado no topo da *Stack*. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor C9H), necessita de 10 ciclos do CPU para a sua execução e não afecta as *flags*.

Após executada, a instrução RET coloca o endereço de retorno no registo PC, *low* (PC) = [SP], *high* (PC) = [SP+1], ficando SP = SP+2.

6.6.7. RETcondição

Instrução de retorno condicional ao programa, após a execução da rotina, se a condição especificada for verdadeira (*RETurn i f "condição" is true*); coloca no registo PC o endereço de retorno, guardado no topo da *Stack* e salta incondicionalmente para o programa. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 6 a 12 ciclos do CPU para a sua execução e não afecta as *flags*.

Após a sua execução, a instrução RET coloca o endereço de retorno no registo PC, *low* (PC) = [SP], *high* (PC) = [SP+1], fazendo SP = SP+2.

Tabela 6.23 – **Instruções de retorno condicional ao programa para várias condições**

Instrução	*Opcode*	*Condição*	*Observações*
RZ	C8	Z = 1	*Return if Zero*
RNZ	C0	Z = 0	*Return if Not Zero*
RC	D8	C = 1	*Return if Carry*
RNC	D0	C = 0	*Return if Not Carry*
RPE	E8	P = 1	*Return if Parity Even* (par)

RPO	E0	P = 0	*Return if Parity Odd* (ímpar)
RM	F8	S = 1	*Return l if Minus* (negativo)
RP	F0	S = 0	*Return if Plus* (positivo)

6.6.8. RST n

Instrução de recomeço (*ReSTart*) incondicional do programa para o endereço 8∗n, sendo n=0,..,7; provoca a reinicialização do CPU. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 12 ciclos do CPU para a sua execução e não afecta as *flags*.

A execução da instrução RST põe o endereço de retorno no topo da *Stack*, [SP]= PC, ficando SP = SP-2, o registo PC recebe o endereço especificado, PC = 8∗n (n=0,..,7).

Tabela 6.24 – **Instruções de recomeço incondicional do programa para vários endereços**

Instrução	*Opcode*	*Address*
RST 0	C7	0000
RST 1	CF	0008
RST 2	D7	0010
RST 3	DF	0018
RST 4	E7	0020
RST 5	EF	0028
RST 6	F7	0030
RST 7	FF	0038

6.7. *INSTRUÇÕES DE ACESSO À STACK*

6.7.1. PUSH r_p

Instrução que coloca o conteúdo do par de registos especificado (*PUSH register par*) no topo da *Stack*, nas células apontadas pelo registo SP. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 12 ciclos do CPU para a sua execução e não afecta as *flags*.

Após executada, a instrução PUSH coloca o conteúdo do par de registos r_p no topo da *Stack*, [SP-1] = r_p, [SP-2] = *pair* (r_p), ficando SP = SP-2.

Tabela 6.25 – ***Opcodes* de PUSH r_p (em hexadecimal)**

r_p / *par* / *[SP]*	*B*	*D*	*H*	*PSW*
	C	*E*	*L*	
	C5	D5	E5	F5

O registo *Processor Status Word* (PSW) corresponde ao par de registos AF – Acumulador e registo de *Flags* –.

6.7.2. POP r_p

Instrução que retira para o par de registos especificado (*POP register par*) o conteúdo das duas células do topo da *Stack*, apontadas pelo registo SP. Esta instrução tem um *byte* de tamanho para o *opcode*, necessita de 10 ciclos do CPU para a sua execução e não afecta as *flags*.

Após executada, a instrução POP coloca o conteúdo do par de células do topo da *Stack* no par de registos indicado, r_p = [SP], *pair* (r_p) = [SP+1], ficando SP = SP+2.

Tabela 6.26 – ***Opcodes* de POP r_p (em hexadecimal)**

r_{dp}	*par*	*[SP]*
B	C	C1
D	E	D1
H	L	E1
PSW		F1

6.7.3. XTHL

Instrução que troca entre si o conteúdo do topo da *Stack* pelo conteúdo do par de registos HL (*eXchange stack Top with H and L*). Esta instrução tem um *byte* de tamanho para o *opcode* (de valor E3H), necessita de 16 ciclos do CPU para a sua execução e não afecta as *flags*.

Após executada, a instrução XTHL coloca o conteúdo do par de células do topo da *Stack* no par de registos indicado e vice-versa, L ⇔ [SP], H ⇔ [SP+1], ficando SP inalterável.

6.7.4. SPHL

Instrução que move para o registo SP o conteúdo do par de registos HL (*SP is moved from HL*). Esta instrução tem um *byte* de tamanho para o *opcode* (de valor F9H), necessita de 6 ciclos do CPU para a sua execução e não afecta as *flags*.

6.8. INSTRUÇÕES DE ENTRADA/SAÍDA

6.8.1. IN port

Instrução que lê o porto de entrada (*INput*), de endereço *port* especificado e transfere o dado de um *byte* para o Acumulador. Esta instrução tem dois *bytes* de tamanho, um para o *opcode* (de valor DBH) e outro para o dado, necessita de 10 ciclos do CPU para a sua execução e não afecta as *flags*.

6.8.2. OUT port

Instrução que escreve no porto de saída (*OUTput*), de endereço *port* especificado o conteúdo do Acumulador. Esta instrução tem dois *bytes* de tamanho, um para o *opcode* (de valor D3H) e outro para o dado, necessita de 10 ciclos do CPU para a sua execução e não afecta as *flags*.

6.9. INSTRUÇÕES DE CONTROLO

Para melhor compreender o funcionamento das instruções de controlo aconselha-se a leitura do capítulo 8, na parte dedicada às interrupções do CPU 8085.

6.9.1. DI

Instrução que desabilita qualquer pedido de interrupção (*Disable Interrupt*) mascarável, desactivando, para isso, a *Flag* de *Enable Interrupt* (FEI = 0). Esta instrução tem um *byte* de tamanho para o *opcode* (de valor F3H), necessita de 4 ciclos do CPU para a sua execução e não afecta as *flags*.

6.9.2. EI

Instrução que habilita um pedido de interrupção (*Enable Interrupt*) mascarável, colocando, para isso, FEI = 1. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor FBH), necessita de 4 ciclos do CPU para a sua execução e não afecta as *flags*.

6.9.3. HLT

Instrução que pára (*HaLT*) o funcionamento do CPU até que haja um pedido de interrupção ou seja activada a linha Reset por acção externa. Após o pedido de interrupção feito, o CPU processa a rotina correspondente ao pedido. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 76H), necessita de 5 ciclos do CPU para a sua execução e não afecta as *flags*.

6.9.4. SIM

Instrução que activa os pedidos de interrupção mascarável (*Set Interrupt Mask*) através do conteúdo binário do Acumulador, em que cada bit controla um pedido de interrupção específico, conforme está indicado na Figura 6.5.

O LSB do Acumulador, *bit* b0, corresponde ao estado da máscara do *interrupt* RST 5.5 (RST 5.5 *Mask*). Quando o estado do *bit* é "1" (*Set*) o *interrupt* está desabilitado ou mascarável. O mesmo sucede com os *bits* b1 e b2 que representam as máscaras RST 6.5 e RST 7.5, respectivamente.

As *interrupts* RST 5.5 e RST 6.5 são *high-level sensitive*, isto é, consideram-se activas quando a "1". A *interrupt* RST 7.5 é *rising-edge sensitive* porque é activa a "1", está mascarada, quando há uma transição ascendente da entrada, funcionando como o *flip-flop D edge-triggered*.

O *bit* b3 quando a "1" habilita o acesso às entradas RST (*Mask Set Enable*). Se b3 está a "0" o estado das máscaras RST (*bits* b0, b1 e b2) não é reflectido na linha de *interrupt*.

O *bit* b4 permite fazer o *reset* ao *interrupt* RST 7.5 *flip-flop* quando no estado "1" (*Reset RST 7.5 flip-flop*).

Os *bits* b6 e b7 destinam-se ao controlo de uma linha de entradas/saídas (I/O) série. O *bit* b7 corresponde ao pino de saída série do CPU (*Serial Output Data, SOD*). O *bit* b6 permite habilitar o sinal SOD (*SOD Enable*).

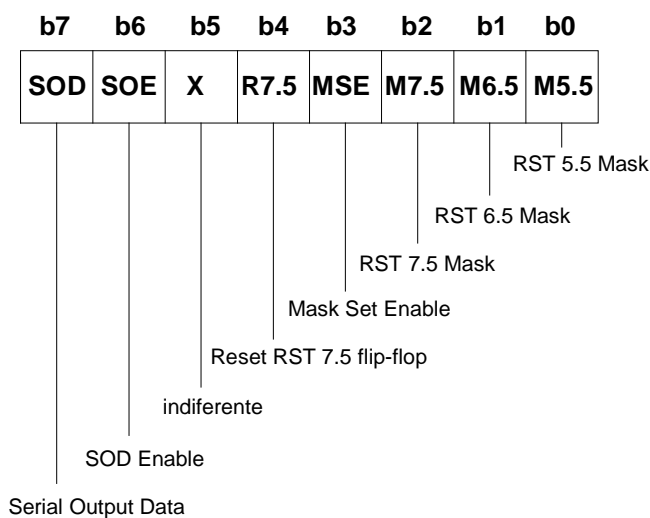



Figura 6.5 – **Função binária do Acumulador antes da execução da instrução SIM**

O *bit* b5 não produz qualquer acção, sendo o seu estado indiferente.

A instrução SIM permite transferir o conteúdo do Acumulador para o *Input Mask Register* do módulo interno de atendimento de *interrupts*. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 30H).

6.9.5. RIM

Instrução que permite transferir o conteúdo do *Output Mask Register* do módulo de atendimento de *interrupts* para o Acumulador (*Read Interrupt Mask*). Após executada a instrução o conteúdo binário do Acumulador tem o seguinte significado:

Os *bits* b0, b1 e b2 do Acumulador habilitam as máscaras de *interrupt* RST 5.5, RST 6.5, RST 7.5 (*RST Mask*), quando a *Set* (estado lógico "1").

O *bit* b3 indica o estado da *Flag Enable Interrupt* (FEI).

Os *bits* b4, b5 e b6 indicam o estado do *Input Mask Register*, correspondente às máscaras RST 5.5, RST 6.5 e RST 7.5, respectivamente.

O *bit* b7 indica o valor presente no pino de entrada série do CPU (*Serial Input Data*, *SID*).

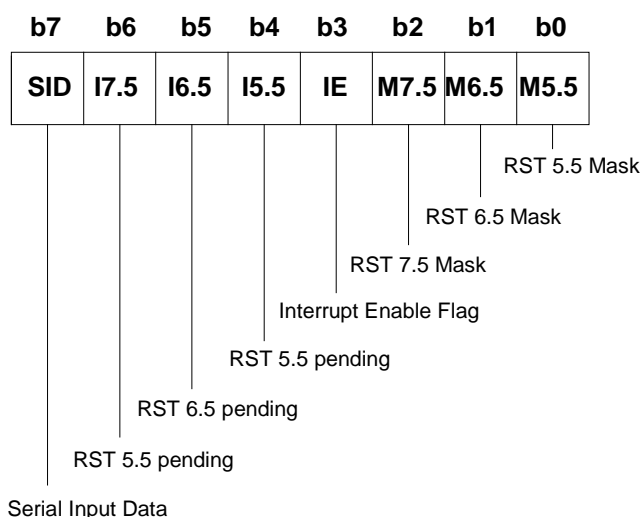



Figura 6.6 – **Função binária do Acumulador após a execução da instrução RIM**

A instrução RIM tem um *byte* de tamanho para o *opcode* (de valor 20H).

6.9.6. NOP

Instrução em que o CPU não realiza nenhuma operação efectiva (*Not OPeration*), provocando apenas o incremento de uma unidade do registo PC. Todavia, esta instrução é utilizada nas rotinas de atendimento de *interrupts* e nas rotinas de *delay*. Esta instrução tem um *byte* de tamanho para o *opcode* (de valor 00H), necessita de 4 ciclos do CPU para a sua execução e não afecta as *flags*.

7. ESTRUTURA INTERNA DO MICROPROCESSADOR 8085

7.1. INTRODUÇÃO

O estudo da arquitectura interna de um microprocessador corresponde a conhecer as características da sua estrutura interna.

Os microprocessadores (μP) podem ser concebidos para aplicações gerais ou específicas. O μP 8085 da Intel é considerado um dos primeiros μP's pois tem uma arquitectura relativamente simples. Este μP é, por isso, utilizado em aplicações didácticas, de introdução aos microprocessadores.

O μP 8085 sucedeu ao μP 8080, tendo beneficiado da evolução tecnológica da integração de componentes: incorporou as principais funções dos integrados 8224 e 8228, necessários no 8080; manteve a compatibilidade ao nível do *software*, tendo melhorado profundamente as interrupções.

Além disso, enquanto o μP 8080 necessita de três fontes de alimentação (-5V, +5V e +12V) o 8085 utiliza apenas uma tensão de +5V. A tabela abaixo sintetiza as características técnicas do μP 8085.

Tabela 7.1 – **Características técnicas do μP 8085**

Característica	*Dado técnico*
Tecnologia	N-MOS em silício
Alimentação	Tensão única de +5V
Capacidade de endereçamento	64 kbytes
Período de relógio	1.3 μs, 8085A; 0.8 μs, 8085A2
Número de instruções	78
Modos de endereçamento	implícito, imediato, directo, por registo directo, por registo indirecto
Interrupções	uma interrupção não mascarável (Trap); vários interrupções mascaráveis de acesso condicionado
Barramento de dados	Multiplexado, *byte* de menor peso do endereço durante o ciclo da máquina τ_D, validação pelo sinal ALE
Terminais (I/O)	Comunicação série SID (*Input*) e SOD(*Output*)
Software	inteiramente compatível com o 8080

7.2. DIAGRAMA INTERNO DO CPU 8085

A arquitectura do microprocessador (μP) 8085 pode ser compreendida através do seu diagrama interno.

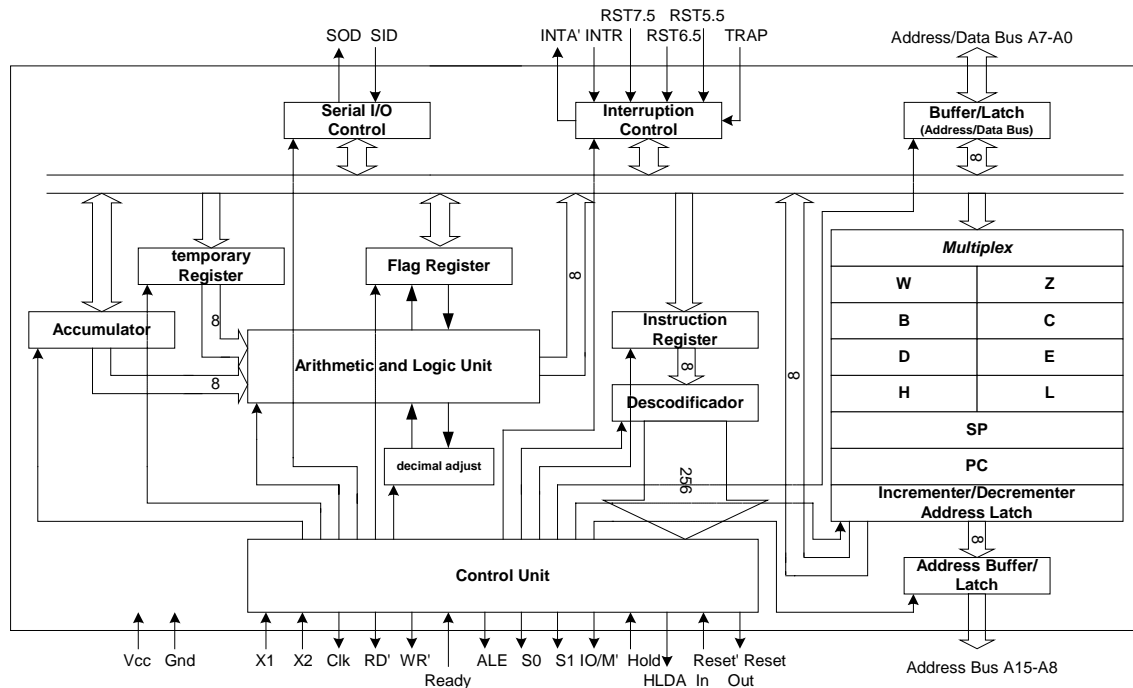



Figura 7.1 – **Diagrama interno do µP 8085**

O µP 8085 permite enviar e receber do exterior dados (*Data*) de 8 *bits*, para isso utilizando o barramento (*Bus*) AD7-AD0, bidireccional e *tri-stade* (*Data Bus*).

O *Bus* AD7-AD0 permite ainda enviar os 8 LSB do endereço. Para que os mesmo acessos possam transmitir dados e endereços é usado um sistema de multiplexagem no tempo. Este mecanismo é gerado pelo sinal de controlo ALE (*Address Latch Enable*) que indica aos dispositivos externos se os 8 *bits* armazenados no *buffer* são dados ou endereços.

Os 8 MSB do barramento de endereços (*Address Bus*) A15-A8 são armazenados no *buffer* de endereços, também *tri-stade*.

Os sinais de controlo utilizam linhas unidireccionais. Estes sinais podem ser enviados pelo CPU aos dispositivos externos ou vice-versa, quando as unidades externas pretendem controlar o funcionamento do CPU.

A estrutura interna do µP 8085 pode ser reduzida a três unidades:

- Unidade de registos, Register Unit (RU) destinada a armazenar os dados e os endereços utilizados na execução de um programa;
- Unidade de controlo, Control Unit (CU) para gerir o funcionamento do CPU;
- Unidade aritmética e lógica, Arithmetic and Logic Unit (ALU) tendo por função realizar as operações aritméticas e lógicas a serem executadas pelo CPU.

O microprocessador 8085 possui dez registos internos:
1º Accumulator (A),
2º Flag register (F) ou *status register*,
3º General registers (B, C, D, E, H e L),
4º Program Counter (PC),
5º Stack Pointer (SP).

Os registos estão agrupados aos pares, permitindo deste modo definir dados de 16 *bits*.

Tabela 7.2 – ***Organização dos registos internos do CPU***

A	*F*
B	*C*
D	*E*
H	*L*
PC	
SP	

7.3. UNIDADE DE REGISTOS

O microprocessador 8085 processa dados de 8 *bits*, pelo que os registos de dados que possui são de 8 *bits*. Mas para o armazenamento dos endereços são usados registos de 16 *bits*, devido ao *Address Bus* ser desse tamanho.

7.3.1. REGISTOS DE USO GERAL

Esta unidade possui 6 registos de 8 bits para o processamento de dados – registos B, C, D, E, H e L.

Estes registos são chamados de General Register porque a sua aplicação não é específica, é de uso geral, tanto são utilizados no processamento de dados como no armazenamento de endereços.

Os registos de uso geral podem ser associados aos pares, nas utilizações BC, DE e HL, permitindo o processamento de dados de 16 *bits*. Nesta situação, os registos B, D e H armazenam o *byte* de maior peso do dado.

Tendo por função armazenar dados a processar ou em processamento, os registos permitem diminuir o número de acessos à memória, aumentando a velocidade de execução dos programas.

O par de registos HL, além de uso geral tem a função específica de funcionar como ponteiro da memória, guardando o endereço da célula de memória a aceder, numa operação de escrita ou de leitura. O registo H guarda o *byte* de maior peso do endereço.

Além destes registos existem ainda os registos W e Z, de 8 *bits*, inacessíveis ao programador mas utilizados pelo CPU para transferir internamente os dados e guardar o operador em operações destrutivas.

7.3.2. PONTEIRO DA PILHA

O registo Stack Pointer (SP) tem 16 *bits* de tamanho e aponta para a memória RAM, numa área designada por *Stack*, permitindo armazenar o conteúdo dos vários registos (BC, DE, HL e PSW), excepto os registos W e Z.

O registo SP aponta sempre ou para o topo da *Stack* ou para o endereço da célula de memória onde foi colocado o último *byte*.

Quando é efectuada uma operação de escrita na *Stack*, o registo SP é decrementado de duas ou três unidades, conforme o operando é de um ou dois *bytes*. Numa operação de leitura o registo SP é incrementado de duas ou três unidades, para operandos de um ou dois *bytes*.

7.3.3. CONTADOR DO PROGRAMA

O registo Program Counter (PC) tem 16 *bits* de tamanho e aponta para a célula de memória onde o CPU deve ler a instrução que vai ser executada, funcionando com o contador do programa a executar.

Na execução de uma instrução que pode ter um, dois ou três *bytes* de tamanho, o registo PC é incrementado automaticamente uma, duas ou três unidades, permitindo ao CPU a leitura do operador e dos possíveis operandos. No final, o registo PC fica a apontar para o *opcode* da próxima instrução.

7.3.4. BUFFER/LATCH DO BARRAMENTO DE DADOS/ENDEREÇOS

O registo Buffer/Latch (Data/Address Bus) tem 8 *bits* de tamanho, constituído por 8 *latchs/buffers* bidireccionais, destinado a armazenar o *byte* de menor peso do endereço ou o *byte* do dado a ser enviado ou recebido. Este registo é acedido pelo *bus* AD7-AD0.

7.3.5. BUFFER/LATCH DO ENDEREÇO

O registo Address Buffer/Latch tem 8 *bits* de tamanho, constituído por 8 *latchs/buffers* unidireccionais, destinado a armazenar o *byte* de maior peso do endereço. Este registo é acedido pelo *bus* A15-A8, cujo conteúdo é enviado para o *Address Bus*.

7.3.6. INCREMENTO/DECREMENTO DO LATCH DE ENDEREÇO

O registo Incrementer/Decrementer tem 16 *bits* de tamanho e permite incrementar/ decrementar o conteúdo presente nos registos:

- *General Registers*, por acção das instruções próprias para o efeito;
- *Stack Pointer* e *Program Counter*, por efeito das acções de acesso à *Stack* ou da execução das instruções.

O registo *Address Latch* tem 16 *bits* de tamanho, constituído por 16 *latch*, utilizado para armazenar o conteúdo dos registos de uso geral que será enviado para o *Address Bus*.

7.4. UNIDADE DE CONTROLO

As instruções executadas pelo CPU são codificadas num dado de um *byte*, chamado *opcode* – código de operação da instrução –.

A execução de uma instrução compreende a realização de várias etapas por parte do CPU:

1ª Leitura do *opcode*, sendo transferido da memória para o registo de instrução (I);

2ª Transferência do *opcode* para o descodificador de instrução que codifica os *clock cycle* necessários para a completa execução da instrução;

3ª Transferência da informação descodificada para a unidade de controlo que gera a execução da instrução.

O registo de instrução, designado por I, serve para o CPU guardar o *opcode*, antes da instrução ser descodificada e, de seguida, executada.

O descodificador de instrução possui 256 saídas, sendo activada apenas uma delas pelo *opcode* recebido. A informação associada a cada saída indica à unidade de controlo o conjunto de actividade que terá de realizar.

A unidade de controlo combina a linha activa, vinda do descodificador, com os impulsos de *clock*, gerando, de modo, o sinal de controlo que actuará sobre os registos, a unidade aritmética e lógica, os *buffers*/*latchs*, as memórias e os dispositivos de I/O. O envio de sinais de controlo às unidades internas e externas do CPU permitirá a execução da instrução.

7.5. UNIDADE ARITMÉTICA E LÓGICA

7.5.1. INTRODUÇÃO

A Arithemetic and Logic Unit (ALU) permite a realização de diversas operações e que são as seguintes:

1ª Operações aritméticas, tais como a adição, a subtracção, o incremento, o decremento, o ajuste decimal, etc.;

2ª Operações lógicas, sendo exemplo o E, o OU, o OU-exclusivo, a comparação, o complemento, etc.;

3ª Operações de rotação, como são a rotação à direita do conteúdo binário do acumulador com a *flag* de *carry*, etc.

A maior parte das operações realizadas pelo ALU utiliza o Acumulador para guardar um dos operandos; outras utilizam-no para armazenar o resultado da operação.

7.5.2. ACUMULADOR

Registo de 8 bits, designado por A, definido como um operando implícito na maior parte dos seguintes tipos de instruções:

- Transferência de dados utilizando a memória;
- Operações aritméticas e lógicas;
- Acesso às unidades de entrada/saída.

7.5.3. REGISTO TEMPORÁRIO

Registo de 8 bits que armazena temporariamente o segundo operando de uma operação aritmética e lógica. O primeiro operando fica armazenado no registo A (Acumulador).

7.5.4. REGISTO DE FLAGS

Registo de 8 bits, designado por F, sendo cinco deles utilizados para armazenar a informação do estado do ALU, reflectida na realização de uma dada operação. Cada um dos cinco bits relevantes do registo F é chamado de Flag – cujo estudo detalhado foi feito no capítulo 4 –.

O par de registos AF constitui o *Program Status Word* (PSW).

Além de fornecer informação sobre o estado de execução do ALU, as *flags* são também utilizadas pelo CPU para a tomada de decisão da sequência das instruções na execução de um programa.

8. ESTRUTURA EXTERNA DO MICROPROCESSADOR 8085

8.1. INTRODUÇÃO

Um microprocessador (μP) corresponde ao Central Processing Unit (CPU) que é implementada num único *chip*. O *chip* do μP 8085 é fabricado na tecnologia MOS de canal N, tendo 40 pinos (contactos externos) para as ligações.

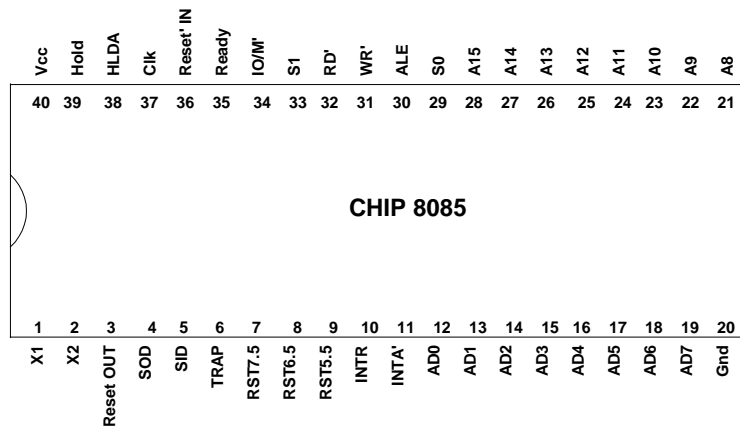



Figura 8.1 – **Configuração externa do μP 8085**

Os pinos do *chip* permitem ao μP a comunicação com o exterior.

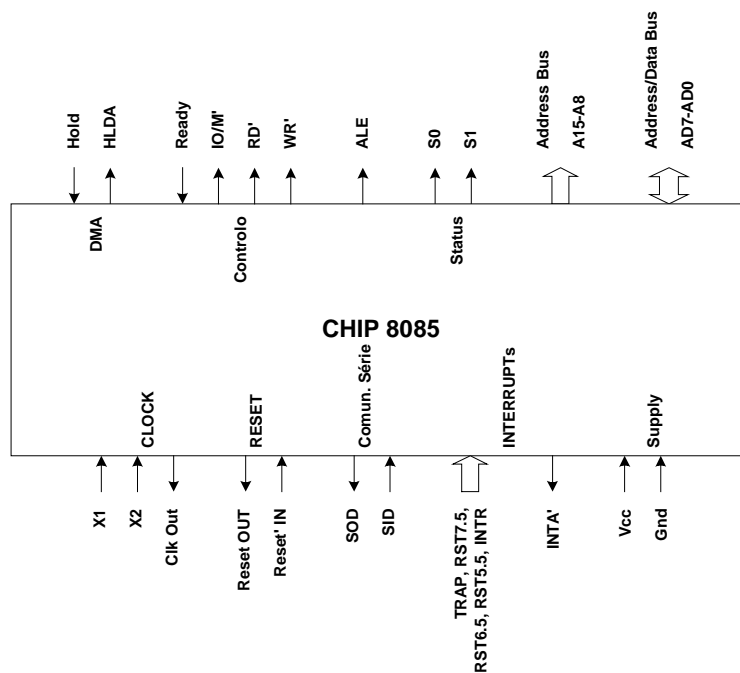



Figura 8.2 – **Fluxo da informação no μP 8085**

Os sinais de comunicação, para cada um dos pinos, podem corresponder à informação dos dados, dos endereços ou do controlo; o fluxo da informação pode ser no sentido da saída ou da entrada.

8.2. ALE

Para realizar a multiplexagem dos pinos AD7-AD0, os quais permitem o acesso ao *Data Bus* e ao *byte* de menor peso do *Address Bus*, o CPU utiliza o sinal Address Latch Enable (ALE).

O processo de multiplexagem é realizado em duas etapas:

1ª Colocação do endereço nos pinos AD7-AD0:
   - 1º O CPU coloca nos pinos AD7-AD0 os 8 LSB do endereço (A7-A0), ao mesmo tempo que coloca o restante endereço nos pinos A15-A8;
   - 2º O sinal ALE é activado (ALE=1) pelo CPU, habilitando a entrada do *Latch*;
   - 3º O *Latch* guarda o byte A7-A0, disponibilizando-o à saída;
   - 4º O CPU desactiva o sinal ALE (ALE=0), desabilitando a entrada do *Latch*.

2ª O CPU pode utilizar os pinos AD7-AD0 como *Data Bus* (D7-D0) pois o sinal ALE está desactivo, não permitindo que o dado passe para o *Address Bus*.

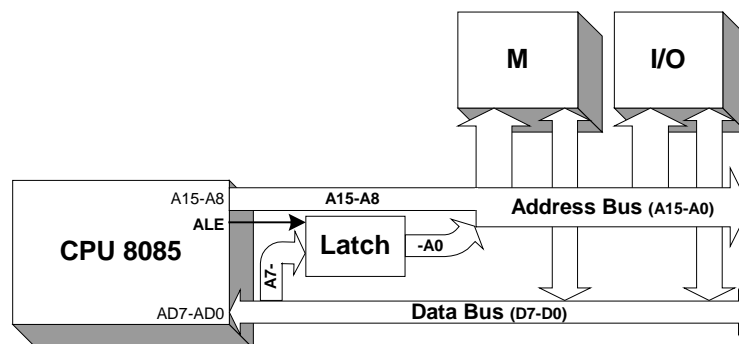



Figura 8.3 – **Multiplexagem do Address/Data Bus no μP 8085**

8.3. SINAIS DE CONTROLO

8.3.1. IO/M’

Sinal de saída *tri-state* que selecciona a unidade de Input/Output ou a unidade de Memória (IO/M) como dispositivo a ser acedido para as operações de leitura ou de escrita.
O sinal IO/M’ quando está no estado lógico “1” selecciona o dispositivo IO; quando a “0” é feito o acesso à memória – sinal activo quando a “0”, M’ –.

A selecção de uma posição de memória ou de uma porta de IO é feita por endereçamento. O sinal IO/M’ permite distinguir uma posição de memória de uma porta de IO, inclusive para o caso de terem o mesmo endereço.

8.3.2. RD’

Sinal de saída *tri-state* que selecciona a operação de leitura (ReaD) aquando do acesso ao dispositivo de Input/Output ou de Memória.

O sinal RD’ é activo quando está a “0”.

8.3.3. WR’

Sinal de saída *tri-state* que selecciona a operação de escrita (WRite) aquando do acesso ao dispositivo de Input/Output ou de Memória.

O sinal WR’ é activo quando está a “0”.

8.3.4. READY

Sinal de entrada, enviado pelo dispositivo de Input/Output ou de memória que indica ao CPU a sua disponibilidade imediata (Ready) para a realização das operações de escrita ou de leitura.

As unidades de memória ou de entrada/saída processam a informação muito mais lentamente do que o CPU. A existência de um sinal que informe o CPU quando pode efectuar uma operação de leitura ou de escrita, aumenta muito a eficiência do funcionamento do microprocessador.

Ao pedido para realizar uma operação de leitura ou de escrita, o sinal *Ready* quando no estado lógico “0” indica ao CPU que ainda não está disponível. Neste caso, o CPU gera estados de espera, correspondendo a *cycles clock* extra, até que o dispositivo a aceder esteja pronto.

O sinal *Ready* quando no estado lógico “1” indica ao CPU que a operação de leitura ou de escrita pode ser feita imediatamente.

A comunicação do CPU com um dispositivo de memória é feita através dos barramentos de dados, de endereços e de controlo, conforme indicado na figura abaixo. O diagrama inclui já os sinais de controlo estudados.

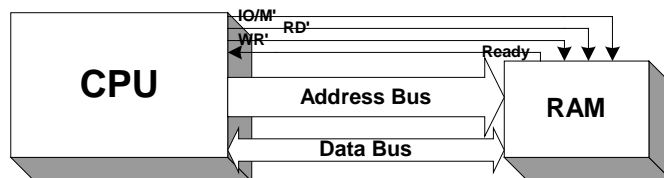



Figura 8.4 – **Interligação do CPU à memória RAM**

8.4. STATUS OUTPUTS

A informação do estado das saídas (Status Outputs) permite analisar o funcionamento do CPU. As ferramentas (tools) de *debugger* do CPU utilizam os sinais de *output* para o teste de falhas ou de *bugs* do programa em execução.

Além dos sinais de saída já conhecidos (IO/M’, RD’, WR’ e Ready), existem mais dois sinais, S0 e S1 que fornecem informação sobre o *status* do Bus do CPU. A interpretação do estado dos cinco sinais completa a informação dos *status* do CPU.

Tabela 8.1 – **Interpretação dos sinais IO/M’, RD’, WR’, Ready, S0 e S1**

IO/M’	*RD’*	*WR’*	*Ready*	*S0*	*S1*	*Status (CPU)*
0	0	1	0	-	-	Aguarda a leitura da memória
0	0	1	1	1	0	Leitura da memória
0	1	0	0	-	-	Aguarda a escrita da memória
0	1	0	1	0	1	Escrita da memória
0	1	1	X	1	1	Pesquisa do código de oeração
1	0	1	0	-	-	Aguarda a leitura da I/O
1	0	1	1	1	0	Leitura da I/O
1	1	0	0	-	-	Aguarda a escrita da I/O
1	1	0	1	0	1	Escrita da I/O
1	1	1	X	1	1	Ocorrência de uma interrupção
HI	0	0	X	0	0	Halt
HI	X	X	X	X	X	Hold
HI	X	X	X	X	X	Reset

NOTA: os estados X e HI correspondem, respectivamente, ao estado irrelevante e ao estado de alta impedância (High Impedance).

O significado dos últimos três estados do CPU é o seguinte:

- O *status* Halt corresponde à paragem do funcionamento do CPU;
- O *status* Hold é utilizado pelas unidades de I/O para acederem directamente à memória;
- O *status* Reset corresponde ao salto do CPU para a posição inicial da memória, apagando o conteúdo do registo de instrução (I) e desactivando todos os *interrupts*.

8.5. ACESSO DIRECTO À MEMÓRIA

Nas operações convencionais de entrada/saída de dados o controlo é feito pelo CPU. Quando se pretende armazenar vários dados introduzidos pelo porto de entrada, o CPU guarda-os um a um numa posição intermédia. Os dados lidos à entrada são colocados no Acumulador e, posteriormente, guardados na memória no endereço especificado.

Existem dispositivos de I/O que permitem transferir dados para a memória sem necessitarem de utilizar o CPU. Esses dispositivos permitem o acesso directo à memória (Direct Memory Access), sendo chamados de Controladores DMA.

Durante uma operação de DMA o CPU não realiza nenhuma operação de acesso às unidades de I/O ou à Memória (IO/M). Essa inibição destina-se a evitar os conflitos de acesso simultâneo do CPU e do DMA às unidade de IO/M.

Para ocorrer uma operação de DMA o controlador solicita ao CPU o controlo temporário do sistema de Bus. Este pedido é feito através do sinal *Hold*.

A inclusão de um controlador DMA num microcomputador torna a eficiência do CPU em operações de acesso às unidade de I/O muito maior.

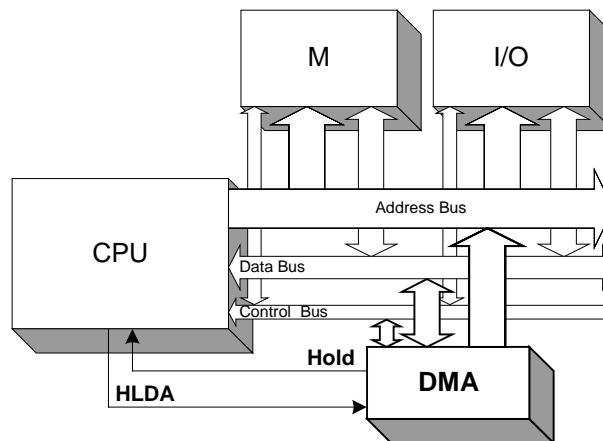



Figura 8.5 – **Diagrama do microcomputador com DMA**

8.5.1. HOLD

Sinal de entrada, enviado pelo controlador de DMA ao CPU, destinado a requisitar (Hold) o controlo das unidades de IO/M e dos barramentos para a transferência directa dos dados.

O processamento interno do CPU pode continuar durante a operação de DMA. As operações realizadas no ALU, desde que apelem apenas à escrita ou leitura de dados entre os registos, podem ocorrer, ainda que o sinal *Hold* esteja activo.

Quando nenhuma operação de DMA ocorre, a entrada *Hold* está no estado lógico “0”.

O CPU, quando recebe um sinal de *Hold*, termina a execução da operação presente. Depois, avisa o DMA que a requisição foi recebida. Para isso:
1º Envia o sinal HLDA para o DMA;
2º Coloca os sinais IO/M’, RD’ e WR’ no estado *tri-state*;
3º Coloca os sinais para o *Data Bus* e o *Address Bus* no estado *tri-state;*
4º E o sistema passa a estar disponível para controlo pelo DMA.

8.5.2. HLDA

Sinal de saída, enviado pelo CPU a indicar a requisição ao DMA (HoLding by DmA) do controlo do sistema, igualmente significando que os *Bus* estão no estado de alta impedância, HI.

Assim que o sinal *Hold* é removido pelo DMA, o sinal *HLDA* é desactivado, passando o CPU a controlar novamente o sistema.

8.6. RESET

O CPU recebe o sinal de *Reset* automaticamente quando é ligado à fonte de alimentação do sistema. Também pode ocorrer manualmente se for accionada a chave de *Reset*.

O CPU possui uma entrada de *Reset* (Reset In) destinada a receber o sinal de *Reset*, vindo de um circuito RC.

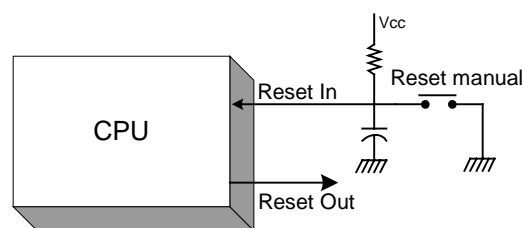



Figura 8.6 – **Esquema do Reset ao CPU**

O circuito RC provoca um atraso no *Reset*, evitando uma transição brusca de *Reset* activo – estado lógico “0” – para Reset desactivo – estado lógico “1” –.

O CPU, pelas características da sua construção, só provoca o *Reset* ao ser ligado ao sistema ou se a entrada *Reset In* permanecer activa – estado lógico “0” – durante um mínimo de 10 ms.

Ao ser activada a entrada *Reset In* do CPU, o *Program Counter* (PC) é colocado no endereço inicial, o conteúdo do registo de instrução é colocado a "0" assim como os seus *flip-flops* internos, enquanto as entradas RST5.5, RST6.5 e RST7.5 são desactivadas. O conteúdo do Acumulador e dos registos de uso geral, assim como o estado das *flags* não são colocados obrigatoriamente a "0"; os seus valores finais são imprevisíveis.

Enquanto a entrada *Reset In* está activa o CPU permanece no estado *Reset*. Neste estado, o *Data Bus*, o *Address Bus*, as linhas de controlo IO/M', RD' e WR' são mantidas a *tri-stade*. Este sinal deve estar activo durante três *clock cycle*.

O sinal de saída *Reset Out* é activado quando a entrada *Reset In* é "0". Este sinal serve para indicar que o CPU está no estado *Reset*. Por isso, também pode ser utilizado para fazer o *Reset* a outros componentes do microcomputador.

A linha de Reset do CPU 8085 destina-se a reiniciar o seu funcionamento e está associada ao endereço 0000H.

8.7. CLOCK

O CPU possui um gerador de *clock* do qual faz parte um circuito ressonante, colocado entre os pinos X1 e X2. O circuito implementado pode ser do tipo RC ou LC. No entanto, é geralmente utilizado um cristal.

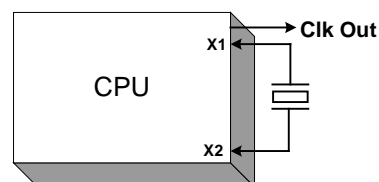



Figura 8.7 – **Circuito gerador do *clock* do CPU**

Para as características típicas do µP 8085-A, a frequência de ressonância do cristal é de 6,144MHz. Internamente, o CPU divide a frequência do cristal em dois, gerando, deste modo, um sinal de *clock* de 3,072MHz ao nível MOS para o processamento das instruções.

O CPU 8085-A fornece aos outros componentes do microcomputador um sinal de *clock* de 3,072MHz, disponível em TTL, através da saída *clock Out*.

O CPU aceita um *clock* gerado externamente, utilizando o pino X1 para o acesso do sinal.

8.8. SINAIS DE INTERRUPÇÃO

8.8.1. INTRODUÇÃO

Quando um sinal externo é enviado ao CPU por um periférico, provocando a paragem da sua actividade, estamos perante uma interrupção (*Interrupt*). O mesmo sinal, acedido numa linha de interrupção, levará o CPU, após interrompido, a executar a rotina de interrupção que foi pedida.

Cada *interrupt* provoca um salto incondicional do programa para uma rotina de atendimento a um pedido de interrupção. Após executar a rotina de *interrupt*, o CPU retoma a execução do programa principal, na instrução imediata à última que tinha sido executada.

O µP 8085 disponibiliza seis *interrupts* que funcionam como sinais de entrada – INTR, RST5.5, RST6.5, RST7.5 e TRAP – e de saída – INTA –.

8.8.2. MASCARÁVEIS E NÃO MASCARÁVEIS

As interrupções podem ser mascaráveis ou não mascaráveis:

- Uma interrupção não mascarável corresponde à aplicação de um sinal de interrupção e à sua inerente resposta por parte do µP, ao pedido de interrupção. Nesta interrupção o CPU responde sempre, qualquer que seja o estado e a execução do programa.
- Uma interrupção mascarável corresponde a uma interrupção que pode ser habilitada ou não (mascarável) por *software*. Nesta interrupção as entradas podem ser mascaradas (desligadas) do CPU com o recurso às instruções específicas – SIM para activar as máscaras, RIM para ler as máscaras –, já vistas anteriormente.

A entrada TRAP é o exemplo duma interrupção não mascarável, devendo o sinal permanecer no estado lógico “1” até que a interrupção seja reconhecida pelo CPU.

As entradas INTR, RST5.5, RST6.5 e RST7.5 são exemplos de interrupções mascaráveis. Quando não estão mascaradas, o sinal de interrupção deve permanecer no estado lógico “1” até que seja reconhecido pelo CPU.

8.8.3. PRIORIDADES

Quando ocorrem vários pedidos de interrupção simultâneos, o CPU impõe prioridades ao atendimento das interrupções.

As interrupções do CPU 8085 têm prioridades de atendimento diferentes e endereços de salto incondicional específicos, à semelhança da instrução RST n. Estes endereços

pertencem à memória ROM e apontam para rotinas correspondentes aos pedidos de *interrupt*.

Tabela 8.2 **– Interrupções do CPU com diferentes prioridades e endereços específicos**

Interrupção	*Prioridade*	*Endereço*	*Observações*
TRAP	1	0024	*Interrupt* não mascarável
RST 7.5	2	003C	*Interrupt* mascarável
RST 6.5	3	00034	Idem
RST 5.5	4	002C	Idem
INTR	5	-	O endereço depende da instrução

8.8.4. INTR

Sinal de entrada, destinado a pedir uma interrupção (INTerrupt Request). O sinal é activado e desactivado (mascarável) por *software*. A desactivação faz-se com o *Reset* e logo depois do atendimento do *interrupt*.

O atendimento do sinal INTR é feito no ciclo de instrução imediato ao do pedido e também nos estado *Hold* e *Halt*. Durante este ciclo o registo PC não é incrementado; poderá ocorrer um *Restart* ou um *Call*, provocando a chamada da rotina de serviço à interrupção.

8.8.5. RST5.5, RST6.5, RST7.5

Sinais de entrada, destinados a gerar um recomeço interno por interrupção (ReSTart interrupt). Estes *interrupts* têm a mesma temporização que o sinal INTR mas têm maior prioridade; são na mesma mascaráveis mas com o recurso à instrução SIM.

8.8.6. TRAP

Sinal de entrada que faz com que o CPU "cai no laço" (TRAP) da interrupção no fim do ciclo da instrução que está em execução. Este *interrupt* é o de maior prioridade, além de ser uma operação não mascarável.

8.8.7. INTA'

Sinal de saída, destinado ao CPU a reconhecer a interrupção (INTerrupt Acknowledge). Este sinal é enviado no ciclo de instrução seguinte ao que aceitou o INTR, substituindo o sinal RD'.

8.9. COMUNICAÇÃO SÉRIE

O *chip* do 8085 corresponde a um microprocessador de 8 *bits*, capaz de receber, processar e enviar dados de um *byte*.

O tratamento de dados de um *byte* corresponde ao processamento paralelo da informação de 8 bits.

O CPU 8085 pode também tratar dados de um *byte* em processamento série. Para isso, o *chip* dispõe de dois pinos destinados à comunicação série de dados, acedidos através de instruções específicas:

- O pino SID (*Serial Input Data*) é acedido através da instrução SID;
- O pino SOD (*Serial Output Data*), acessível com a instrução SOD.

9. TEMPORIZAÇÕES DO MICROCOMPUTADOR 8085

9.1. INTRODUÇÃO

Um microprocessador opera ciclicamente. A execução de um programa resume-se à realização de uma sequência de operações de leitura e de escrita, de acesso à memória e às unidades de *Input/Output* (I/O), correspondendo a operações de transferência de um byte, seja de dado, de endereço ou de código de instrução (*opcode*).

As operações *Read* (RD) e *Write* (WR) constituem as únicas formas de comunicação do CPU com as outras unidades mas também são suficientes para a execução de qualquer instrução do programa.

O período necessário para a realização do ciclo de busca (*cycle for fetch*) e do ciclo de execução (*cycle for execute*) de uma instrução designa-se de Ciclo de Instrução. Este período varia de instrução para instrução.

As etapas correspondentes ao ciclo de instrução, executadas pelo CPU, são as seguintes:

1º Leitura do *opcode* na memória;

2º O registo de instrução guarda o *opcode*;

3º Descodificação da instrução e envio para a unidade de controlo;

4º Execução da instrução, transformada numa activação de sinais de controlo com uma sequência bem definida.

Se a instrução executada tiver operandos que são dados, o CPU coloca o segundo operando no registo temporário. Se um dos operandos é um endereço, o CPU coloca-o directamente no *Address Bus*.

O CPU 8085 tem um *Data Bus* de 8 *bits*, pelo que só podem ser processados dados de um *byte* e um de cada vez.

As operações de escrita ou de leitura efectuadas pelo CPU decorrem durante um período designado por Ciclo de Máquina. Cada ciclo de instrução pode ter entre um a cinco ciclos de máquina.

O período de busca da instrução necessita de um ciclo de máquina por cada *byte* da instrução. Cada ciclo de máquina corresponde, por sua vez, entre três a seis *clock cycles*.

9.2. CICLOS DE MÁQUINA

Os ciclos de máquina correspondem a cinco tipos diferentes de operações, identificados pelos sinais IO/M', RD' e WR' do CPU 8085.

Tabela 9.1 – **Identificação dos tipos de ciclos de máquina**

Ciclos de máquina	*IO/M'*	*RD'*	*WR'*
Busca de *opcode*	0	0	1
Leitura de memória	0	0	1
Escrita de memória	0	1	0
Leitura de I/O	1	0	1
Escrita de I/O	1	1	0

9.3. EXEMPLIFICAÇÃO

Para exemplificar o processo da temporização na execução de uma instrução pelo CPU, considere-se o caso da instrução STA addr – armazenamento do conteúdo do Acumulador na célula de memória de endereço addr –.

A instrução "STA addr" tem três *bytes* de tamanho – um para o *opcode* e dois para o endereço (addr) –. A sua execução corresponde à realização de três ciclos de máquina para a leitura dos três *bytes* da memória e a um ciclo de máquina para a escrita do conteúdo do Acumulador na posição de memória especificada.

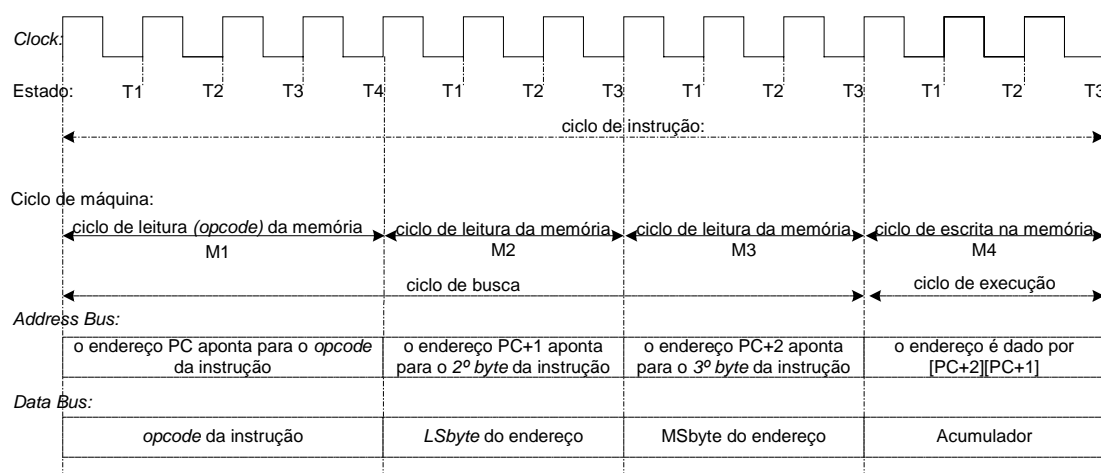



Figura 9.1 – **Ciclo de instrução para STA addr**

Durante a execução do primeiro ciclo de máquina, o CPU realiza as seguintes operações:

1º Coloca o conteúdo do registo *Program Counter* (PC) no *Address Bus*, ocupando para isso, o *clock cycle* T1;

2º Efectua a leitura do *opcode* na célula de endereço PC e armazena-o no registo de instrução (I), utilizando o *clock cycle* T2;

3º Descodifica o *opcode*, identificando a instrução STA e reconhecendo que serão necessários mais dois ciclos de máquina para ler o endereço e um para escrever na memória, para isto tendo utilizado o *clock cycle* T3;

4º Incrementa o registo PC para aceder à próxima célula de memória – PC=PC+1 –, ocupando o *clock cycle* T4.

Os dois ciclos de máquina seguintes permitem ao CPU ler o endereço da célula de memória a aceder para a operação de escrita, armazenando-o no par de registos W-Z, realizando as seguintes operações:

I.1º Coloca o conteúdo do registo *Program Counter* (PC) no *Address Bus*, ocupando para isso, o *clock cycle* T1;

I.2º Efectua a leitura do *LSByte* na célula de endereço PC actual e armazena-o no registo W, utilizando o *clock cycle* T2;

I.3º Incrementa o registo PC para aceder à próxima célula de memória – PC=PC+1 –, ocupando o *clock cycle* T3.

II.1º Coloca o conteúdo do registo *Program Counter* (PC) no *Address Bus*, ocupando para isso, o *clock cycle* T1;

II.2º Efectua a leitura do *LSByte* na célula de endereço PC actual e armazena-o no registo Z, utilizando o *clock cycle* T2;

II.3º Incrementa o registo PC para aceder à próxima célula de memória – PC=PC+1 –, ocupando o *clock cycle* T3.

No último ciclo de máquina, o CPU realiza a execução propriamente dita da instrução:

1º Coloca no *Address Bus* o endereço da célula a aceder, durante o *clock cycle* T1;

2º Coloca no *Data Bus* o conteúdo do Acumulador, no *clock cycle* T2;

3º Escreve na célula de endereço addr a informação contida no *Data Bus* – supondo que a memória está disponível –, durante o *clock cycle* T3.

9.3.1. BUSCA DE OPCODE

O ciclo de busca de *opcode* utiliza quatro a seis *clock cycle* do CPU para a sua execução. Neste ciclo, o µP 8085 tem de ler, armazenar e descodificar a informação do *opcode* para poder processar a instrução.

No ciclo de busca do *opcode*, constituído por seis *clock cycles*, o CPU realiza as seguintes acções:

1º Estado T1 – o sinal de controlo é configurado para o ciclo de busca do *opcode*, IO/M’=0, sendo os sinais de *status* activados, S0=1 e S1=1;

– é enviado o conteúdo do registo PC para o *Address B*us que selecciona a célula de memória que contém o *opcode* da instrução (o *byte* mais significativo do endereço, PCH, é colocado nas linhas A15-A8, enquanto que o *byte* menos significativo, PCL, é colocado nas linhas AD7-AD0);

– o *byte* menos significativo, PCL, é passado para as linhas A7-A0 quando o sinal ALE é activo;

– o conteúdo das linhas AD

2º Coloca no *Address Bus* o endereço da célula a aceder, durante o *clock cycle* T1;

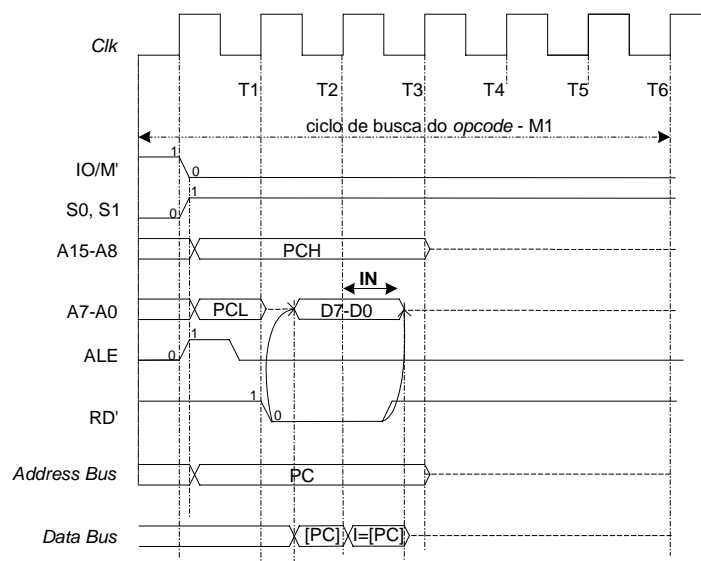



Figura 9.2 – **Ciclo de busca do *opcode* com seis *clock cycle***

10. ESTRUTURA DA MEMÓRIA DO MICROCOMPUTADOR 8085

10.1. INTRODUÇÃO

O estudo da arquitectura interna de um microprocessador corresponde a conhecer as características da sua estrutura interna.

10.2. INSTRUÇÕES DE ACESSO À STACK

O CPU armazena na memória o conteúdo do registo PC, sempre que é atendido um pedido de *Interrupt* ou é chamada uma rotina. Este armazenamento temporário é realizado numa parte da memória RAM designada por Stack.

O *Stack* corresponde a um empilhamento de células de memória, definidas num espaço localizado no topo da memória RAM. O acesso ao *Stack* é realizado a partir da célula de maior endereço. O CPU define um Stack Pointer (SP) que é um ponteiro que aponta sempre para o topo da pilha que está disponível para armazenamento.

A figura seguinte ilustra a organização da memória, classificando-a pelas componentes ROM, RAM e Stack.

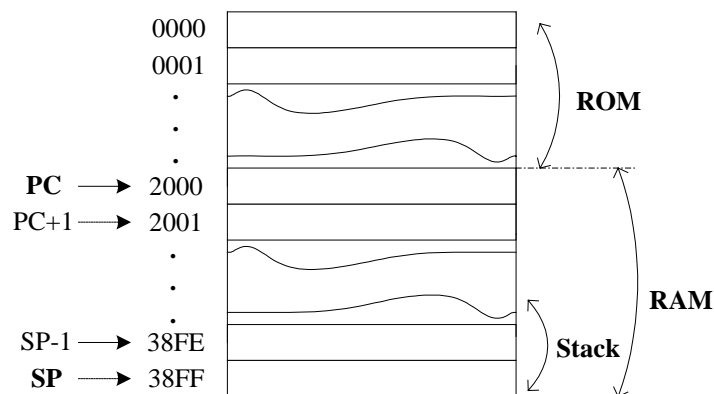



Figura 10.1 – **Organização da memória ROM, RAM e da *Stack***

É através do registo SP que o CPU acede ao *Stack*. Quando é realizada uma operação de escrita no *Stack*, o registo SP é decrementado uma unidade, enquanto que numa operação de leitura o SP é incrementado em uma unidade.

O registo SP aponta sempre para o topo do *Stack*. Nos endereços abaixo do SP as células de memória estão disponíveis para operações de escrita; nos endereços iguais ou superiores ao SP encontram-se as células de memória que guardam a informação resultante da operação de escrita.

A transferência de dados entre um programa e o *Stack* é feita segundo as seguintes regras:

1ª. Guardar um dado de 16 *bits* no *Stack* equivale a colocar os 8 MSBs na célula de endereço SP-1 e os 8 LSBs na célula de endereço SP-2, ficando SP = SP-2;

2ª. Colocar num par de registos o conteúdo de duas células do *Stack* adjacentes, equivale a colocar o conteúdo da célula de endereço SP no registo de menor peso e o conteúdo da célula de endereço SP+1 no registo de maior peso, ficando SP = SP+2.

Sendo o primeiro valor a ser colocado no Stack o último valor a ser retirado, diz-se que a sua organização é do tipo LIFO, *the Last In is the Firt Out*.

Quando o CPU atende um *interrupt* ou é feita uma chamada a uma rotina as primeiras operações a serem realizadas são as seguintes:

- [SP-1] = *high*(PC), guarda o *byte* de maior peso do endereço de retorno no topo do *Stack;*
- [SP-2] = *low*(PC), guarda o *byte* de menor peso do endereço de retorno;
- SP = SP-2, o registo SP fica a apontar para o topo do *Stack* não escrito;
- PC = addr, o registo PC guarda o endereço de salto incondicional do CPU para o *Interrupt* ou a rotina a executar.

O *Stack* é também utilizado para o armazenamento temporário de dados. Neste caso o acesso ao *Stack* é feito através de instruções específicas:

- PUSH r_p: instrução que escreve no topo do *Stack* o conteúdo do par de registos r_p;
- POP r_p: instrução que lê do topo do *Stack* o conteúdo de duas células de memória e escreve no par de registos r_p.

Para a execução da instrução PUSH r_p são realizadas as seguintes operações:

- [SP-1] = *high*(r_p),
- [SP-2] = *low*(r_p),
- SP = SP-2.

Para a execução da instrução POP r_p as operações a realizar são as seguintes:

- *low*(r_p) = [SP],
- *high*(r_p) = [SP+1],
- SP = SP+2.

As instruções PUSH e POP são utilizadas no acesso às rotinas e ao atendimento de *interrupts*. Estas instruções permitem que os registos utilizados nas rotinas e que guardam dados relativos ao programa principal, tenham o seu conteúdo preservado para futuras operações.

Para preservar os valores dos registos que irão ser utilizados na rotina, no início devem ser executados os PUSHs que colocam o seu conteúdo no *Stack*. Para recuperar os conteúdos originais dos registos utilizados, antes do fim da rotina devem ser executadas os POPs.

A ordem com que as instruções POPs são executadas numa rotina deve ser a inversa da ordem com que as instruções PUSHs apareceram para que os valores recuperados sejam os correctos e porque o *Stack* é do tipo FIFO, *the Firt In is the Last Out*.

10.3. INTERRUPÇÕES DO CPU

Uma interrupção, também chamadas de Interrupt, provoca a paragem do funcionamento do CPU aquando da execução do programa corrente, para iniciar a execução da rotina de atendimento de *Interrupt*.

Um pedido de interrupção (*Interrupt ReQuest*) é feito por um dispositivo externo ao CPU, através da activação de uma das cinco entradas de *Interrupt*. As linhas de *Interrupt* podem interromperem qualquer instante a actividade presente do CPU, direccionando-o para as rotinas de atendimento dos dispositivos que solicitam a interrupção.

Ao receber um *Interrupt Request* o CPU termina a execução da instrução que se encontrava a processar, respondendo ao IRQ no *cycle clock* imediato. A primeira instrução que executa permite guardar o conteúdo do registo PC na *Stack*. A segunda instrução coloca no registo PC o endereço associado à entrada de *Interrupt*. Só depois executa a rotina de atendimento de *Interrupt*.

Os endereços dos *Interrupts* correspondem a células de memória ROM, onde está guardado o programa monitor. Nestes endereços existem instruções de salto incondicional para os endereços de memória RAM, onde está definida a rotina de atendimento de *Interrupt* que foi seleccionada.

Após executada a rotina de atendimento de *Interrupt*, o registo PC fica a apontar para a instrução imediatamente a seguir à última instrução executada do programa, quando ocorreu a interrupção.

A sequência de acções realizadas pelo CPU antes e depois da execução de uma rotina de atendimento de *Interrupt* é designada de Vector Interrupt.